Secure Execution of Mobile Agents on Open Networks Using Cooperative Agents

YU Chiu-Man

A Thesis Submitted in Partial Fulfilment

of the Requirements for the Degree of

Master of Philosophy

in

Computer Science and Engineering

August 2002

Abstract

Security is an important issue in mobile agent systems. A mobile agent can be easily attacked by malicious hosts, especially if it is executed on open networks where there is no trusted third party. The works presented in this thesis focus on the execution and state tampering problem. Since mobile agents are executed on remote hosts, malicious hosts can tamper with program states of the agents during execution. It is difficult to prevent this tampering attack as the computing resources are provided by the remote hosts.

We have developed a tamper-detection protocol to protect the execution of mobile agents by using cooperating agents. There are two kinds of cooperating agents in the protocol: a coordinator to coordinate the tamper-detection processes; and a detector to perform re-execution on remote hosts. They continuously monitor the mobile agent and detect any malicious hosts along the mobile agent's itinerary. The detection mechanism of our protocol is derived from the execution trace approach with code obfuscation. The detector contains an obfuscated program which is a transformed version of the mobile agent's program. The mobile agent leaves an execution trace when it is executed on a remote host. The detector uses the trace to perform re-execution and sends the detection information back to the coordinator. The coordinator then checks if there is any tampering attack. Compared with existing protocols, our tamper-detection protocol can preserve the asynchronous execution characteristics of mobile agents, and is flexible since hosts need only to provide data storage service in order to deploy the protocol. Experimental results show that the overall overheads of the protocol is acceptable. Logical verifications conclude that the message transmissions of the protocol is secure.

We have also developed several extensions to our protocol. One of the extensions can increase the security strength of the protocol. An extension deploying program slicing techniques can speed up the overall detection process. Another extension enables the protocol to support multiple identical mobile agents. All the extensions require certain trade-offs.

流動代理人在開放網絡上的安全執行〔利用合作代理人〕

撮要

保安是流動代理人系統中的一個重要課題。流動代理人很容易被惡意主機攻擊，尤其是當它在沒有可靠第三者的開放網絡上執行。本論文中所述之研究工作針對執行及狀態竄改的問題。因爲流動代理人在遠端主機上被執行，惡意主機可以在執行中竄改代理人的程式狀態。因爲運算資源由遠端主機提供，防止這種竄改攻擊是因難的。
我們利用合作代理人發展了一種竄改偵察協定去保護流動代理人的執行。在協定中有兩種合作代理人：一種是協調者，它會協調各種竄改偵察工作；另一種是偵察者，它會在遠端主機上運行「再執行」。它們在流動代理人的路途中持續監察流動代理人及偵察任何惡意主機。我們的協定中的偵察機制是演化自執行追蹤技術加上代碼混淆技術。偵察者含有一個由流動代理人程式轉變出來的混淆化程式。當流動代理人在遠端主機上被執行時，它會留下一段執行追蹤。偵察者利用這段追蹤去運行「再執行」，並將偵察資訊傳送給協調者。協調者然後檢查有否竄改攻擊的跡象。相對現有的協定，我們的竄改偵察協定能保留流動代理人的非同步執行特性，而這協定亦很有彈性，因爲主機們只須提供資料儲存服務便能使用這個協定。實驗顯示這協定的整體耗用時間是可以接受的。邏輯驗證結論出這協定中的訊息傳送是安全的。我們亦爲這協定發展了數個伸延。其中一個伸延可以提昇這協定的安全程度。一個使用程式分割技術的伸延可以加速整體偵察工作。另一個伸延令這協定可以支援數個相同的流動代理人。所有伸延都需要某些取捨。

Acknowledgements

First of all, I am very grateful to my research supervisor, Professor NG Kam-Wing, who has provided numerous helpful guidelines and brilliant suggestions to me throughout my master of philosophy programme.

I would also like to thank Professor LEUNG Ho-Fung and Professor LYU Rung Tsong Michael for their reviews and constructive responses to my thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Mobile agents are software agents with the ability to travel through a network. The term "mobile" means the ability to migrate across the network by hopping from platforms to platforms. The term "agent" has no exact definition. We usually describe agents by their attributes. Generally, agents are software entities with the following attributes [19]: active, autonomous, goal-driven, typically acting on behalf of a user or another agent. A mobile agent can migrate from its home platform to another platform with the appropriate agent environment [32]. The agent's program will be executed on the remote platform rather than on the home platform. The migrated mobile agent can also migrate to other platforms across the network. Figure 1.1 shows the migration characteristics of a mobile agent.

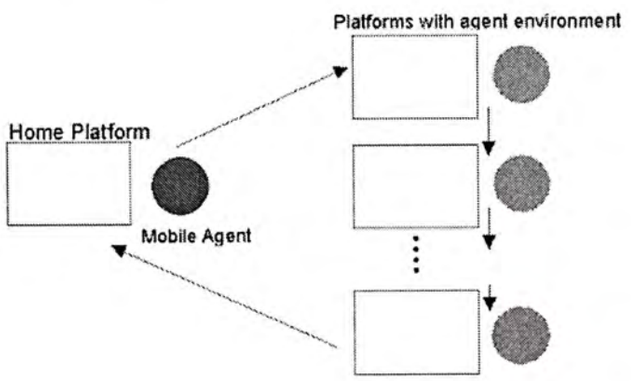

Figure 1.1: Migration characteristics of a mobile agent.

A mobile agent can interact with the components or resources on the platforms if it gets their approvals. It can also communicate with other

agents on a platform if the platform's agent environment supports multiple agents.

The mobile agent paradigm possesses many advantages (see Section 1.1 for details); however, security is one of the major obstacles that prevent the large-scale deployment of mobile agent systems. Since the codes of mobile agents are executed on remote platforms rather than only on the home platform, security concerns arise to protect the agents if the remote platforms are malicious. On the other hand, there are threats to security if the mobile agent is malicious. It may attack the hosts (platforms) which enable it to execute. The security issues will be discussed in later sections.

Comparing to traditional distributed systems (e.g. RPC, CORBA), the mobile agent system faces more security problems due to its agent migration ability. Nevertheless, the problems have to be solved in order to enable mobile agent systems as a feasible alternative to traditional distributed systems.

Mobile agent security issues are mainly divided into "host security" and "agent security" issues. Among the security issues of mobile agent systems, "protecting agents from malicious hosts" is considered as a difficult problem [18]. This is because a mobile agent's processes are executed at remote hosts. It is difficult to prevent the host from analyzing the processes [2]. As a result, the host may be able to alter or modify the processes. Solving this problem is a critical task to the mobile agent paradigm.

1.1 Advantages of mobile agents

A mobile agent has advantages related to its mobility and agent properties [8] [19]. Significant advantages include the following:

Reduced communication bandwidth : A mobile agent is sent to a remote host and performs its task there. After completing the task, the agent (or only the results) is sent back to the home host. The communication cost is sending and receiving the agent. If a large amount of remote server data is needed for processing and the required results contain only little information, the communication cost can be reduced by sending a mobile agent to the data instead of sending the data to the local host.

Customization : Agents are easier to be customized than servers. It is not necessary to install a specific procedure at a server to handle specific client service request. Instead mobile agents can be customized to user needs, and sent to the server where customized requests are executed. This results in more dynamics.

Asynchronous task execution : After the home host has sent a mobile agent out, the home host can perform its own jobs. Simultaneously, the agent is performing tasks at remote hosts. The home host does not need to take care of the agent until the agent comes back.

These advantages makes the mobile agent paradigm very attractive especially in systems where network bandwidth is low or network connection cost is high; and for applications where remote data to be processed is vast or the task to be performed is time-consuming.

1.2 Security

Security is an important issue for mobile agent applications, especially for electronic commerce. We divide the security issues into two sub-issues: host security and agent security.

Host security refers to problems about malicious mobile agents. They may migrate to remote hosts and intrude, for example, changing system settings, steal private information, destroy important information of the hosts, etc. Trojan Horses are well-known attack programs [9]. Since the hosts may continuously accept agents and execute them, one may keep sending mobile agents to explore security loopholes of the hosts. The host security problems also exist in other mobile code systems, therefore they have been well-known since the early times. Popular solution approaches include authentication, authorization and verification of code integrity.

Agent security refers to problems about malicious hosts. Since the agent is executed in the environment provided by remote hosts, the hosts can take complete control of the code, states and data of the agent. If the system is closed and all hosts are trusted, the agent should be safe. However, if the system is open and the host is not trusted, the host may be malicious and attack the agent. [31] [10] illustrate a number of possible attacks. Major attacks include: unauthorized modification of code, extraction of valuable data, denial of execution, and execution tampering. The agent security problem only occurs in mobile agent systems. Our research focuses on solving the problem of execution tampering.

1.3 Contributions

This thesis focuses on the problem of state (or execution) tampering to mobile agents by malicious hosts on open networks. Our contributions can be summarized as the followings:

1. We have developed a tamper-detection protocol (refer to Chapter 5) to solve the problem. Compared with existing protocols, our protocol is more flexible and suitable on open networks where no trusted third party exists. Moreover, mobile agents pay only little overhead when using the protocol.

2. We have developed several extensions to the protocol (refer to Chapter 8, 9, 10, 11) to enhance the protocol in certain aspects. The aspects include security, efficiency and scalability. Reasonable trade-offs are needed to deploy the extensions.

3. We have verified the protocol using a formal logic, BAN logic (refer to Chapter 6). From the verification, we can conclude that message transmissions of the protocol are secure.

4. We have conducted some experiments to evaluate the protocol (refer to Chapter 7). We can observe the performance of the protocol from the experiments.

1.4 Structure

The thesis is organized as follows. In Chapter 2 we describe the problem of tampering attack. In Chapter 3 we present several existing approaches to protect mobile agents from malicious hosts. In Chapter 4 we give a brief description about techniques related to our proposed protocol, for example, code obfuscation. In Chapter 5 we describe and discuss the proposed security protocol. In Chapter 6 we verify the protocol using formal logic. In Chapter 7 we present some experiment results of the protocol. In Chapters 8, 9, 10, 11, we describe several extensions which can enhance the protocol in certain aspects with some trade-offs. We conclude the thesis in Chapter 12.

Chapter 2

The Problem of Execution Tampering Attack

2.1 Mobile agent execution model

In this paper, we assume that mobile agents (MAs) possess the execution model as shown in Figure 2.1:

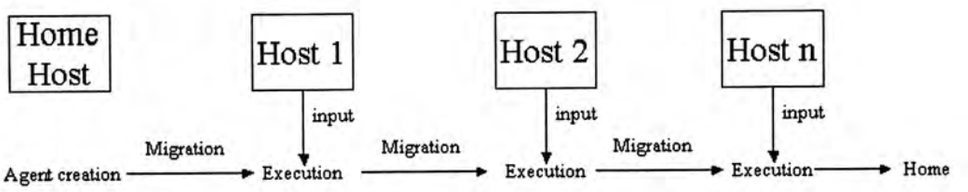

Figure 2.1: Mobile agent execution model.

A MA is created at its home host. It consists of code, data, and program states. To perform tasks, the agent will migrate to $\{host_1, host_2, ..., host_n\}$. On each host, the MA's code is executed. Upon execution, with MA's initial states and input from the host, MA's result states are generated. The result states will become the initial states on the next host. The input represents all data outside the MA. When the MA finishes execution on the final host, it migrates back to the home host.

2.2 Tampering attack from malicious hosts

When a MA migrates to a host, its code, data, and program states are under the host's control. A malicious host may try to alter the MA's code, data

or states to benefit itself [27]. Therefore we need mechanisms to prevent or detect tampering with MA from malicious hosts. Modification of an agent's code, and thus the subsequent behavior of the agent on other platforms, can be detected by having the original author digitally sign the agent's code. Detecting malicious changes to an agent's state during its execution or the data an agent has produced while visiting a compromised platform does not yet have a general solution [13].

For example, a shopping MA has a variable $< visited\_sites >$ to store the list of seller website addresses it has visited in the current journey. If the MA is executed on a malicious seller host, the seller host may append its competitors' website addresses to $< visited\_sites >$ during execution of the MA. In this way, the host suppresses the MA from visiting its competitors. Since the states of the MA have to be loaded to the host for MA's execution, the malicious host can gain access to the states directly. Moreover, the host may tamper with MA's states as mentioned in our example.

2.3 Open network environment

A popular example of an open network environment is the Internet. On an open network, the environments and settings of the hosts can be different from others. The network is exposed to other hosts outside the network. And the open network is dynamic since hosts can join or detach from the network any time. The mobility and automation characteristics of mobile agents are suitable for application on open networks. Mobile agents can perform tasks on hosts of various functionalities, e.g., sales, consultation, trading, etc.

Therefore, we propose a flexible tamper-detection protocol for mobile agents which is suitable for open networks. The protocol does not require a trusted third party. This means that no additional server is required to deploy the protocol. In addition, since hosts need not to install services to support the protocol, it is flexible for hosts to deploy the protocol. The hosts only need to provide data storage for mobile agents, and enable them to communicate with mobile agents on other hosts.

2.4 Conclusion

Execution tampering is an important problem in mobile agent security. The problem does not exist in traditional distributed systems but mobile agent systems. This is because mobile agents are autonomous entities which are executed on remote hosts to perform tasks. If their executions are tampered by

malicious hosts, they will perform the tasks incorrectly. And some malicious hosts may benefit from this.

Mobile agents are particularly suitable on open networks like the Internet due to their mobility and asynchronous execution characteristics. We propose a flexible tamper-detection protocol to protect mobile agents on open networks. The protocol detects if the execution and program state of a mobile agent have been tampered by a malicious host. Deployment of the protocol needs no additional server and service installations on hosts. Therefore it is flexible and suitable on open networks.

Chapter 3

Existing Approaches to Solve the Execution Tampering Problem

3.1 Introduction

To solve the execution tampering problem, there are three main broad approaches in protecting agents from malicious hosts [25]:

Trusted execution environment : This approach ensures that the execution environment for the agent to execute is known to be safe. In this way, the agent needs not to worry about any attacks. A closed system is an example. All hosts in the system are trusted. The utilization of a trusted hardware computing base can also provide a trusted execution environment [8].

Tamper-detection : This approach is to detect whether a mobile agent has been attacked along its journey. This is usually done by checking code integrity, result ranges, execution time, etc. Cryptographic execution trace technique can be used to keep track of the agent's execution along its journey [30].

Tamper-prevention : The object of this approach is to make tampering of agents difficult. To achieve the goal, many different techniques have been suggested. For example, blackbox technique[10], mess-up algorithms, introducing noise code [21], co-operating agents [24], etc.

3.2 Trusted execution environment

3.2.1 Closed system

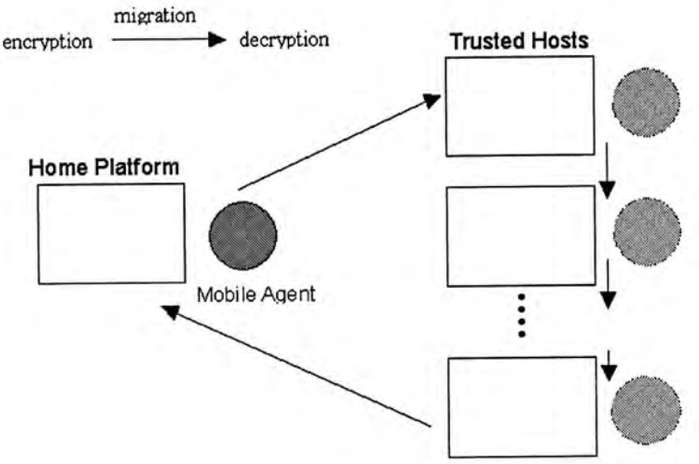

Figure 3.1: Closed system.

It is an effective way to ensure the security of a mobile agent system if the system is built as a closed system. This means all hosts in the system are well-known to each other and are trusted that agents are safe to be executed on them. The mobile agents only migrate to these trusted hosts. Since it is sure that no host is malicious, there is no need to protect the agents. To exclude network-level attack, the agents may be encrypted during migration from hosts to hosts. Figure 3.1 shows an illustration of a closed system.

Unfortunately, many mobile agent systems are not closed system. For example, a very potential application of mobile agents is shopping agent. A user sends a shopping agent to the Internet to shop in virtual stores. It is not likely for the user's host and other virtual stores to form a trusted network. Moreover, open systems are more and more important nowadays due to its extensibility and flexibility. Therefore, we have to study other security approaches that work in open systems.

3.2.2 Trusted hardware

There are some studies of using hardware to provide a trusted computing base in mobile agent systems [33] [8]. The idea is to install hardware devices on the hosts to execute mobile agents. Since hardware cannot be modified by software attack, it should be safe to execute agents on the trusted hardware

even if the hosts are malicious. Since the agents are executed in the devices which are not controlled by the hosts, the hosts can not read the code and data of the agents if the agents are encrypted.

Stefan Funfrocken [8] described using Java Card to protect mobile agents. The Java Card is a card with a processor which offers a Java virtual machine. Therefore the card can execute the code of a mobile agent if the code is written in Java. The cards are installed on the hosts which want to provide an agent environment for correspondent mobile agents. They act as trusted computing bases for the agents. This means executions of the agents are not performed on the hosts' processor but on the Java cards.

To protect mobile agents, the system has to meet some criteria:

- The card must support encryption and decryption.
- There has to be a certification authority.
- The code parts inside the card must be managed.

When an agent migrates to a host with a certified card, it sends its encrypted code and data to the card. The code and data are encrypted because the host may be malicious. The card needs to decrypt them and execute the code. The card should be issued by a certification authority so that it possesses the key to decrypt the code. After the execution is finished, the card encrypts the code and data, and sends them back to the agent. Figure 3.2 shows the mechanism. By this mechanism, even if the host steals the agent's code and data, it cannot decrypt them without the correct key.

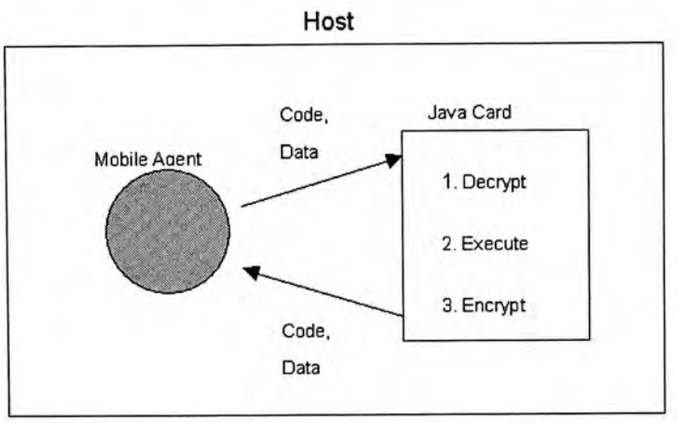

Figure 3.2: Mechanism of using Java Card as trusted computing base.

This approach can ensure high security of the mobile agent system. However, it has weaknesses similar to the closed system approach. It is harsh for all hosts to have to install the trusted hardware in an open system. This limits the number of hosts the agents can travel to even if the hosts are not malicious.

The processing power of trusted hardware is also critical. It is obvious that a host's computing power should be higher than the add-on trusted hardware (e.g. Java Card). This induces that the agents are forced to have worse performance.

3.3 Tamper-detection

3.3.1 Execution tracing

Giovanni Vigna [30] described an approach to detect tampering based on execution tracing and cryptography. The approach is to trace the execution of a migrating agent. Such traces can be used as a basis for problem execution verification, i.e. for checking the agent program against a supposed history of its execution. If there is tampering, cryptographic tracing allows the agent owner to prove that the operations the agent is accounted for could never have been performed.

There are some limitations on this approach. First, the execution trace could be very large even if compressed. It is proportional to the code complexity and the number of hops of the agent in a journey.

Second, the proposed approach can only deal with single threaded agents only. In reality, programs may be multi-threaded. An extension of this approach is needed to solve this problem.

Fritz Hohl [11] developed a protocol based on the trace approach. The basic idea of the protocol is to modify the trace approach such that the execution of a mobile agent on a previous host is checked on the next host in every session. The checking process deploys the re-execution mechanism described in [30]. Figure 3.3 shows the re-execution mechanism of Hohl's protocol.

In Hohl's protocol, the performance of the MA is weakened since every host needs to perform re-execution to check the MA's previous computation before executing the MA. The MA is basically idle until checking is finished. Second, since the tamper-detection process is performed by remote hosts, the hosts need to install services to run the detection process. If the hosts are already supporting environment of a mobile agent system, then the infras-

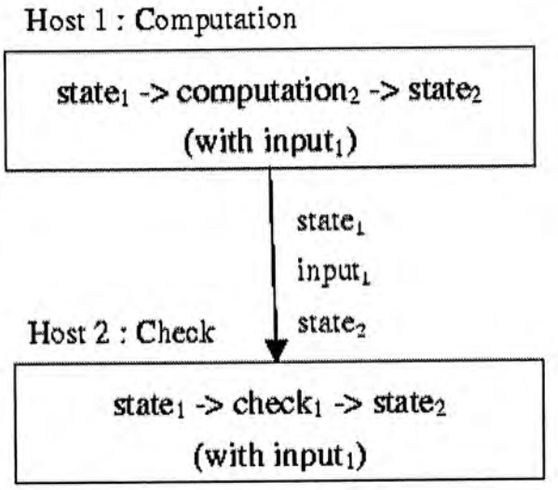

Figure 3.3: Re-execution mechanism in Hohl's protocol.

tructure of the system needs to be modified accordingly in order to deploy the protocol. This is not flexible to implement the protocol on old systems.

In this paper, we propose a new security protocol which can overcome the above problems while providing fast tamper-detection for the MA system.

3.4 Tamper-prevention

3.4.1 Blackbox security

The idea of blackbox security [10] is to generate an agent abstraction in which the code cannot be read by the host even during execution. An agent is a blackbox if the code and data of the agent specification cannot be read nor modified at any time.

This approach is similar to cryptography, but the target is to hide the code specification rather than the data. Tomas Sander and Christian Tschudin described the protocol as computing with encrypted functions and data. With the protocol, an agent specification can be first converted into an encrypted executable agent. When the agent migrates to a remote host, the host needs to execute the agent according to the protocol. Since the agent is encrypted, the host cannot read the agent directly. Figure 3.4 illustrates the blackbox approach.

Unfortunately, there is a fatal restriction of this approach: currently only polynomial and rational functions can be used for this approach. This is also the restriction of the encrypted functions computing protocol. Therefore,

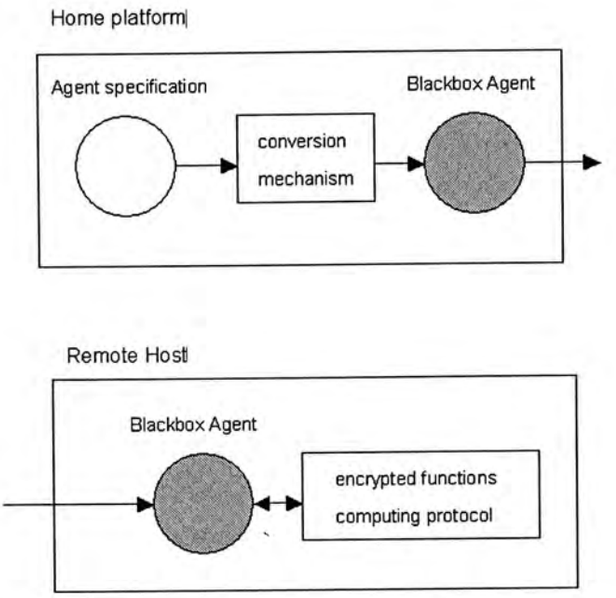

Figure 3.4: Mechanism of using Java Card as trusted computing base.

programs are needed to be checked before using this approach. However, it is very rare to see programs which consist only of polynomial and rational functions. This restriction makes the approach unrealistic.

If the restriction can be overcome, this approach has the advantage that the protection of agent is easily provable and the costs of the protection are probably small.

3.4.2 Time limited blackbox

Fritz Hohl [10] proposed the time limited blackbox approch based on the idea of a blackbox. Since a permanent blackbox cannot be achieved currently, the time limited blackbox approach is proposed instead.

The property of a time limited blackbox is defined as:

an agent is a blackbox if:
For a certain known time interval

1. code and data of the agent specification cannot be read
2. code and data of the agent specification cannot be modified
3. attacks after the protection interval are possible, but these attacks do not have effects

The key in this approach is the introduction of an expiration date. When an agent migrates to a host, an expiration date is calculated and stored. This approach assumes that the host is not able to modify the agent before the expiration date, i.e. the agent is non-modified before the date. When the agent hops again to another host, the new host compares the current date with the expiration date. If the expiration date has not passed, it can assume that the agent has no problem, i.e. valid. Otherwise, the agent may have been modified by a malicious host.

Agent mess-up algorithms are used to emulate a time limited blackbox. The task of a mess-up algorithm is to generate a new agent out of an original agent, which differs in code and data representation but have the same results. In this way, a host cannot directly read the agent specification unless it is able to rearrange the agent into its original form. It is possible for the host to spy on the code successfully. However, an amount of time is needed to do it. Therefore, the expiration date is set accordingly and a time limited black box is generated in this way. Figure 3.5 illustrates the time limited black box approach.

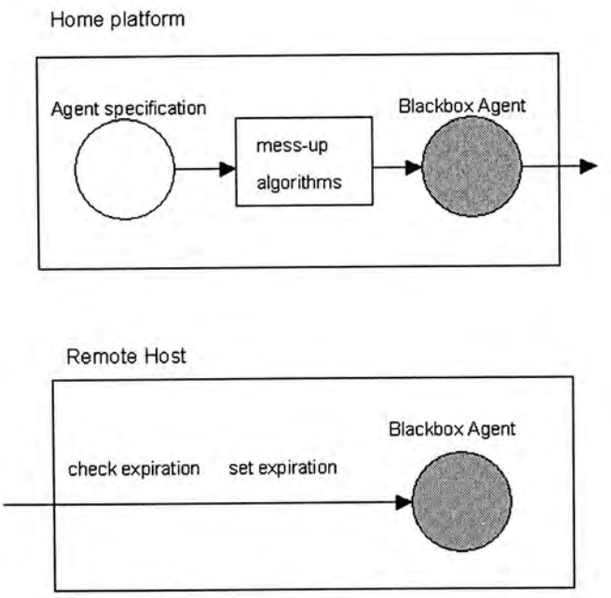

Figure 3.5: Time limited blackbox approach.

There are some drawbacks to this approach. First, it is not easy to determine appropriate expiration dates. The protection intervals have to be long enough for the agent to finish its tasks. However if the period is too long, the agent may be attacked successfully by a malicious host. It is also difficult to predict how much time a malicious host needs to attack an agent. Second,

in order to compare the expiration dates with current dates, synchronized clocks are necessary. Third, the agent needs to perform mess-up again after some period because a host may obtain the original agent specification from the blackbox agent after the expiration time, and then send the information to other hosts.

3.4.3 Agent mess-up

As mentioned in Section 3.4.2, mess-up algorithms can be used to generate a new agent out of an original agent so that they are different in code and data representation but produce the same results. This can increase the attack costs if a malicious host has already known the original agent specification. Since the code organization is changed, the host needs to analyze the agent again if it wants to modify the code.

There are various mess-up algorithms. Three examples [10] are variable re-composition, conversion of control flow elements into value-dependent jumps, and deposited keys.Agent mess-up is not difficult to implement. Java also supports code mess-up. The protection ability can be adjusted by choosing different mess-up algorithms. However, counter attacks exist against the algorithms. They can work effectively if the choice of mess-up algorithm is known. Therefore using multiple different algorithms can provide stronger protection.

3.4.4 Addition of noisy code

Adding noisy code [21] could also increase the attack cost of a malicious host. This approach is to disrupt the malicious host when it tries to locate important code of the agent. The following is a simple example:

original code

```
int new_price;
new_price = get_price();
if (new_price > max_price)
    max_price = new_price;
```

after adding noisy code

```
int new_price;
int new_price2;
new_price = get_price();
new_price2 = get_price();
if (new_price > max_price)
```

```
    max_price = new_price;
if (new_price2 > max_price2)
    max_price2 =new_price2;
```

The example is a program segment that compares a new price with the maximum price. It simply adds a set of dummy code similar to the original code. If a malicious host wants to modify the price comparing code, it will not be sure of the exact code location. The host may take a guess. The probability of a correct guess depends on the amount of dummy code. The probability in the example is 0.5. The more the dummy codes, the lower the probability.

This approach has some drawbacks. First, the noisy code (dummy code) would decrease efficiency of the agent by increasing computation time and resources. The penalty increases if stronger protection is to be achieved. Second, a malicious host may simply modify all the codes which are likely to perform important tasks. In this situation, this approach has no protection effect.

3.4.5 Co-operating agents

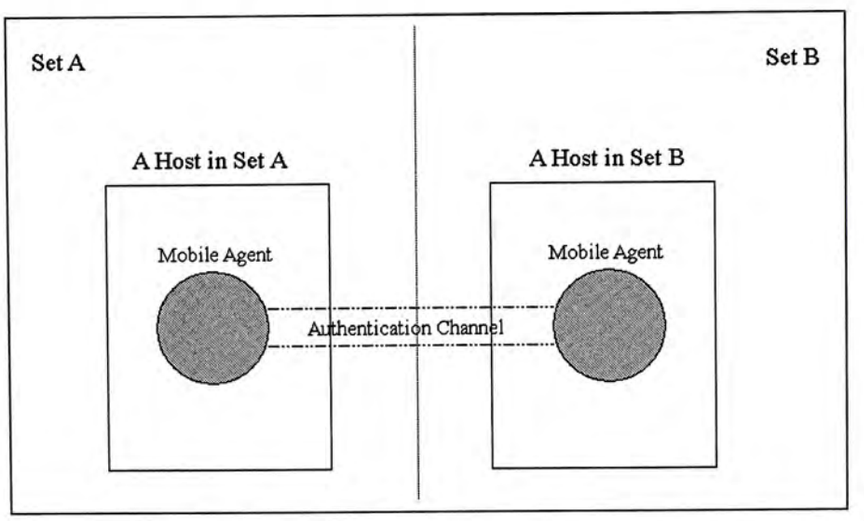

Figure 3.6: Co-operating agents approach.

Volker Roth [24] proposed an approach to use mutual co-operating agents to protect each other from malicious hosts during recording of itineraries. The general idea is to split critical tasks and data between two co-operating agents. When an agent migrates to a host and is to execute, it needs to set up an authenticated communication with the co-operating agent first. The

co-operating agent would perform some critical tasks for its partner and keep monitoring the other's itinerary (Figure 3.6). In this way, no single agent can perform all the tasks alone. The approach sets up two conditions to ensure that the agents are under protection:

- The two agents travel on two non-intersecting sets of hosts
- No malicious host of either set is willing to co-operate with a malicious host of the other set

With these conditions, even if an agent is tampered by a malicious host, the agent cannot function alone. And the co-operating agent can detect the attack if the attack causes the agent migrating in a wrong route (the approach was proposed to record itineraries).

This approach has some drawbacks. First, it is costly in terms of operation time to set up the authenticated communication channel for each migration. Second, the approach has not handled the case if one of the agents is killed. Third, the condition of no cross-set co-operation of malicious host is difficult to prove or verify. Fourth, the decomposition of critical tasks may not be easy to be done automatically. This greatly limits deployment of this approach. Fifth, an agent may be modified by a malicious host in a way that the co-operating agent cannot detect.

3.5 Conclusion

There are three main approaches to solve the execution tampering problem: trusted execution environment, tamper-detection, and tamper-prevention. Trusted execution environment approach aims to ensure safe execution environment in the first place. This approach needs closed network or tamper-proof hardware. Tamper-detection aims to detect if the executions or states of mobile agents are tampered. Vigna's execution tracing [30] is a popular technique but the detection is performed when the mobile agents are back to the home host. Our tamper-detection protocol is built upon the execution tracing technique but detection is performed just after a mobile agent finishes execution on a host. Tamper-prevention aims to prevent mobile agents from execution tampering. Popular techniques used are code cryptography and code obfuscation. However, both techniques cannot prevent unintentional or random tampering by malicious hosts.

Chapter 4

Tamper-Detection Mechanism of Our Protocol

4.1 Introduction

In our tamper-detection protocol, the detection mechanism needs to deploy two computing techniques. They are execution tracing and code obfuscation. Execution tracing is used to make a record of the information of the execution steps. Code obfuscation is used to transform a piece of code into one which is functionally identical but with different program statements. The detection mechanism in our protocol uses an original program to generate an execution trace and uses its corresponding obfuscated program to perform a re-execution. The rationale of this approach will be described in Chapter 5 which shows our tamper-detection protocol. In this chapter, we will describe how the tamper-detection mechanism can be achieved.

4.2 Execution tracing

In Section 3.3.1, we have briefly introduced Giovanni Vigna's approach based on execution tracing to detect tampering. In our tamper-detection protocol, the detection mechanism is based on execution tracing and code obfuscation. We will describe code obfuscation in Section 4.3.

The basic idea of execution tracing is to record information of an execution. The information is called an execution trace. In Giovanni Vigna's execution tracing approach, the trace is composed of a sequence of pairs of $< n, s >$ where n represents a unique identifier of a statement, and s is a signature [29]. A code segment is composed of a sequence of statements. For statements which change the values of internal variables by inputs of external

variables, there are corresponding signatures in execution trace. The signatures contain the information of the inputs of external variables. We show an example in the following.

A mobile agent reads price $price\_i$ on shop $shop\_i$ and compares the price $price\_i$ with the best price $best\_price$ it can get. The code segment shows the sequence of statements and corresponding identifiers.

code segment (pseudo code)of the mobile agent:
1 : connect database on $shop\_i$
2 : read $price\_i$ from database
3 : if $price\_i$ is less than $best\_price$
3.1 : $best\_price = price\_i$
3.2 : $best\_shop = shop\_i$
4 : set $shop\_i$ to next shop in shop list
5 : go $shop\_i$

Assume that there is no tampering, after execution we can obtain the following trace:

execution trace without tampering:
(Assume that the initial value of $best\_price$ is 8.)
1 :
2 : $price\_i = 10$
3 :
4 :
5 :
(Result value of $best\_price$ is 8.)

The execution trace shows the sequence of execution. The numbers represent the identifier of program statements. In Statement 2 of execution trace, the statement reads a value of external variable. Therefore the signature part shows the input value. Statements 3.1 and 3.2 do not appear in execution trace since $price\_i$ is not less than $best\_price$.

By the trace, re-execution is performed in the following sequence:
(Assume that the initial value of $best\_price$ is 8.)
1 :
2 : $price\_i = 10$
3 :

4 :
5 :
(Result value of $best\_price$ is 8.)

Since the result state of original execution and that of re-execution are the same, we can conclude that there is no tampering to the agent's execution.

On the other hand, we assume that the host is malicious this time. The malicious host tampers with the agent's execution by changing the value of $best\_price$ into 100. We will get the following execution trace.

execution trace with tampering
(Assume that the initial value of $best\_price$ is 8.)
1 :
2 : $price\_i = 10$
($best\_price$ is tampered into 100)
3 :
3.1 :
3.2 :
4 :
5 :
(Result value of $best\_price$ is 10.)
Since value of $best\_price$ is tampered into 100 during execution, $price\_i$ is less than $best\_price$ when executing Statement 3. Therefore statements 3.1 and 3.2 appears in execution trace.

By the trace, re-execution is performed in the following sequence:
(Assume that the initial value of $best\_price$ is 8.)
1 :
2 : $price\_i = 10$
3 :
4 :
5 :
(Result value of $best\_price$ is 8.)

In re-execution, the value of $best\_price$ is not tampered. Therefore $price\_i$ is not less than $best\_price$ when executing Statement 3. And the result value of $best\_price$ is 8 not 10. Since the result states of original execution and that of re-execution are different, we can conclude that there is tampering to the agent's execution.

In the example, we have demonstrated how to detect tampering using execution tracing. If there is tampering to the agent's execution, we can observe a difference between the result states of the agent's execution and that of re-execution. On the other hand, if there is no tampering, the result states should be the same.

4.3 Code obfuscation

The purpose of code obfuscation is to transform a piece of code into one that is functionally identical to the original but which is much more difficult for human or machine to learn. A code obfuscator is based on the application of code transformations, in many cases similar to those used by compiler optimizers. We can evaluate the transformations with respect to their potency (the degree a human reader is confused), resilience (the ability to resist automatic deobfuscation attacks), and cost (the overhead added to the application) [6].

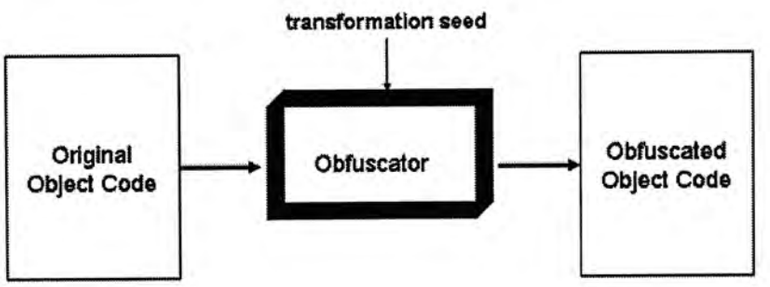

Figure 4.1: Obfuscation of object code.

Figure 4.1 shows an overview of obfuscating a piece of object code into an obfuscated object code. The object code can be a program written in programming languages like C/C++ or Java. Due to different structures of different languages, the obfuscator is dependent on the language of the object code; that is, a program written in Java must use a Java obfuscator. The obfuscator performs obfuscating transformation on the original object code and generates a piece of obfuscated code. Figure 4.2 is an example of obfuscating transformation provided in [6]. It uses loop blocking transformation to generate a piece of code which is more difficult to understand than the original one. There are many other methods for obfuscating transformation. An obfuscator usually can perform several different transformations. [6] provides a detailed survey on this issue. The transformation seed usually

is a random generated number. It is used to control the transformation parameter values and pattern. With different seeds an obfuscator can generate different obfuscated codes from a single object code.

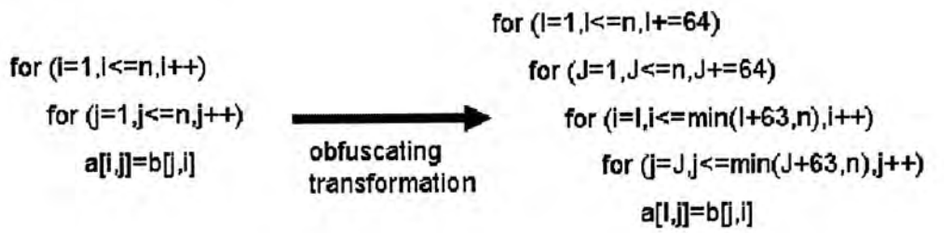

Figure 4.2: An example of obfuscating transformation.

In our tamper-detection protocol, code obfuscation is deployed to confuse the hosts during execution trace checking. The details are described in Chapter 5. Since the target to confuse is the remote hosts, an obfuscator with high resilience is suitable for our protocol.

4.3.1 Resilience of obfuscating transformation

Resilience of an obfuscating transformation indicates the transformation's ability to confuse an automatic deobfuscator. An important factor to resilience is deobfuscator effort. Deobfuscator effort represents the execution time and space required by an automatic obfuscator to reduce potency of the transformation. It can be classified as either polynomial time or exponential time [6].

In our tamper-detection protocol, obfuscated code is used to confuse malicious host during re-execution process. Therefore our protocol requires obfuscating transformations which have high resilience such that deobfuscator effort is exponential time. In this way, a malicious host needs to take a very long time to perform deobfuscation. This is similar to data encryption. And our protocol assumes that a host is malicious if the re-execution process takes too long a time. By selecting suitable obfuscating transformation, a malicious host cannot bypass tamper-detection with deobfuscation.

The complexity of structures of an obfuscated program can affect the time for deobfuscation. An obfuscated program composed of more control structures and variables usually requires more time to be deobfuscated. We assume that the programs of mobile agents are composed of numerous control structures and variables. As a result, there are many permutations for obfuscating transformations to obfuscate the programs. If the programs of

mobile agents have very few control structures and variables, the permutations of obfuscating transformations would become much fewer accordingly. And a malicious host can deobfuscate the program using only little effort.

4.4 Execution tracing with obfuscated program

In our tamper-detection protocol, execution tracing is a component of the detection mechanism. However, we do not directly use the traditional tracing mentioned in Section 4.2. In our detection mechanism, an original program is executed and the corresponding trace is generated. In the re-execution process, we use an obfuscated program (transformed from the original program) instead of the original program. The statements in the obfuscated program can be different from that in the original program. But the obfuscated program and the original program are functionally identical. Their sequences to read external variable inputs are the same. Therefore re-execution by the obfuscated program can still use the trace from execution of the original program.

However, since the statements of the obfuscated code are different from that of the original code and the number of variables of the two codes can be different, the identities of states of the obfuscated program are different from that of the original program. The obfuscated program and original program are the executables generated from the obfuscated code and the original code respectively. The identities of states are the names of variables in the programs. To detect tampering by execution tracing using obfuscated code, we need to generate an "identity mapping" table during code obfuscation and compile-time. The "identity mapping" table records pairs of identities of corresponding states in the obfuscated program and the original program. Using the table, we can compare the states of the obfuscated program to that of the original program.

In the following we show an example.

Original code segment

1 : read($price\_i$)
//read $price\_i$ from external variable input
2 : if ($price\_i < best\_price$)
{
2.1 : $best\_price = price\_i$
2.2 : $best\_shop = shop\_i$

}

Obfuscated code segment

a : $n = 10$
b : $m = 0$
c : read($pricei$)
//read $pricei$ from external variable input
d : $m = pricei$
e : if (($pricei >$ -10) and ($pricei + m < 0$))
{
e.1 : $n = 10$
e.2 : $m = 0$
}
else
e.3 : if ($pricei < bestprice$)
{
e.3.1 : $bestprice = pricei$
e.3.2 : $bestshop = shopi$
}
}

Identity mapping table	
Obfuscated program	Original program
identity of $pricei$	identity of $price\_i$
identity of $bestprice$	identity of $best\_price$
identity of $bestshop$	identity of $best\_shop$
identity of $shopi$	identity of $shop\_i$
identity of n	
identity of m	

The obfuscating transformation used is the insertion of irrelevant controls. The original code and the obfuscated code are functionally identical. For the sake of readability, we change the variable names in the obfuscated code which are different from that in the original code. An "identity mapping" table is generated to record the identities of state pairs. Since the number of variables in obfuscated program is usually more than that in original program, there are records in the table which have identities in the obfuscated program only.

Assume that there is no tampering, after execution we can obtain the following trace:

execution trace without tampering:
(We assume the following initial values: $price\_i = 0$; $best\_price = 8$; $best\_shop$ = "A", $shop\_i$ = "B".)
1 : $price\_i = 10$
2 :
(Result values: $price\_i = 10$; $best\_price = 8$; $best\_shop$ = "A", $shop\_i$ = "B".)

By the trace, re-execution is performed in the following sequence:
(The program loads the following initial values: $price\_i = 0$; $best\_price = 8$; $best\_shop$ = "A", $shop\_i$ = "B".)
a :
b :
c : $pricei = 10$
d :
e :
e.3 :
(Result values: $price\_i = 10$; $best\_price = 8$; $best\_shop$ = "A", $shop\_i$ = "B".)

Checking table			
	Identity mapping table		
Values of identities of obfuscated program		Values of identities of original program	
Value	Identity	Identity	Value
10	identity of $pricei$	identity of $price\_i$	10
8	identity of $bestprice$	identity of $best\_price$	8
"A"	identity of $bestshop$	identity of $best\_shop$	"A"
"B"	identity of $shopi$	identity of $shop\_i$	"B"
10	identity of n		
10	identity of m		

In the re-execution process, the obfuscated program fetches the external variable inputs according to the order of external variable inputs in the execution trace. The values of results states resulted from executions of the original program and that of the obfuscated program (re-execution) are merged with the "identity mapping" table. The combined table is called a checking table. It is used to check if the values of the result states from the original execution are coherent to that from re-execution. In the above case, the values of all identity pairs are matched. Therefore the checking result shows that there is no tampering.

On the other hand, we assume that the host is malicious this time. The malicious

host tampers with the agent's execution by changing the value of $best\_price$ into 100, similar to the example in Section 4.2. We will get the following execution trace.

execution trace with tampering
(We assume the following initial values: $price\_i = 0$; $best\_price = 8$; $best\_shop$ = "A", $shop\_i$ = "B".)
1 : $price\_i = 10$
($best\_price$ is tampered into 100)
2 :
2.1 :
2.2 :
(Result values: $price\_i = 10$; $best\_price = 10$; $best\_shop$ = "B", $shop\_i$ = "B".)
Since value of $best\_price$ is tampered into 100 during execution, $price\_i$ is less than $best\_price$ when executing Statement 2. Therefore statements 2.1 and 2.2 appears in execution trace. The value of $best\_price$ becomes that of $price\_i$.

By the trace, re-execution is performed in the following sequence:
(The program loads the following initial values: $price\_i = 0$; $best\_price = 8$; $best\_shop$ = "A", $shop\_i$ = "B".)
a :
b :
c : $pricei = 10$
d :
e :
e.3 :
(Result values: $price\_i = 10$; $best\_price = 8$; $best\_shop$ = "A", $shop\_i$ = "B".)

Checking table			
	Identity mapping table		
Values of identities of obfuscated program		Values of identities of original program	
Value	Identity	Identity	Value
10	identity of $pricei$	identity of $price\_i$	10
10	identity of $bestprice$	identity of $best\_price$	8
"B"	identity of $bestshop$	identity of $best\_shop$	"A"
"B"	identity of $shopi$	identity of $shop\_i$	"B"
10	identity of n		
10	identity of m		

The checking table shows that two identity pairs have incoherent (different)

values. Therefore the checking result concludes that there is tampering to executions of the programs.

In the example, we have shown how to perform tamper-detection using execution tracing with obfuscated program used in re-execution. An "identity mapping" table is generated during code-obfuscation and compile-time to record the corresponding identities of states in the original program and the obfuscated program. The table is used to check the values of the result states from the original execution with that from re-execution. If the values of all identity pairs are matched, we can conclude that there is no tampering to the executions; otherwise, there is tampering to the executions. The detection mechanism of our tamper-detection protocol deploys this approach.

4.5 Conclusion

Execution tracing is used to make a record of the information during an execution. The information is called an execution trace.It contains the order of executed statements and the corresponding process parameters. Re-execution with execution trace can detect if the execution performed has been tampered.

Obfuscation is to generate a piece of obfuscated code from an original code. The obfuscated code is functionally identical to the original code but more difficult to learn/read. Our protocol deploys code obfuscation techniques to confuse a malicious host. A piece of obfuscated code is used for the re-execution process. There are various obfuscating transformations. Our protocol requires transformations which have high resilience such that the deobfuscator effort requires exponential time. As a result, a malicious host needs to take very long time to break the obfuscated code.

In our tamper-detection protocol, execution tracing and code obfuscation are important components of the detection mechanism. In the detection mechanism, we use an obfuscted program to perform re-execution while the execution trace is resulted from the execution of its original program. To achieve this approach, we need to generate an "identity mapping" table to map the identities of states of the obfuscated program to that of the original program. Based on this table, we can compare and check their result states.

Chapter 5

A Flexible Tamper-Detection Protocol by Using Cooperating Agents

5.1 Introduction

Our aim is to keep detecting if a mobile agent has been tampered by a malicious host along the agent's journey. Once a malicious host is detected, the home host will be informed that the mobile agent is likely to be tampered. Although this cannot prevent tampering, a fast detection can prevent the tampered agent from performing damaging functions.

The detection part of our protocol is carried out by two cooperating mobile agents: a coordinator and a detector. The coordinator is an agent to coordinate the tamper-detection process. It keeps communicating with the mobile agent. When the mobile agent completes execution on a host, the coordinator will send a detector to the host. The detector detects if the mobile agent has been tampered. The detector is an agent which has the ability to detect tampering. In our protocol, the execution tracing approach (mentioned in Section 3.1) with code obfuscation is used for detection. The detector contains the obfuscated MA's code. The detector first loads initial states of the obfuscated code from the coordinator. Then it migrates to a suspecting host to perform detection. The obfuscated code is executed with input stored by MA previously, and generates its result states. The detector compares the difference between result states of MA's trace and that of obfuscated code's trace. The difference information is encrypted and sent back to the coordinator. The coordinator checks if the difference is appropriate. If the result shows that the mobile agent has been tampered,

the coordinator will inform the home host.

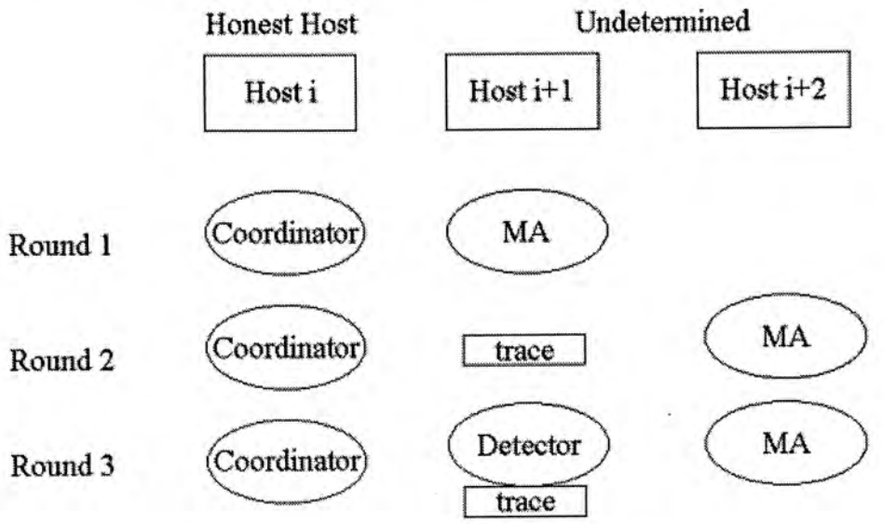

Figure 5.1: Basic flow of the protocol.

Figure 5.1 shows the basic flow of the protocol. The coordinator always situates on honest hosts. A host is honest if it has been checked by the detector and the result shows that it has not tampered with the mobile agent. A host can also be considered honest if it is a trusted host. The mobile agent does not need to do extra job for security except sending its host position to the coordinator and generating the execution trace. When the coordinator finds that the mobile agent has completed execution on a host (change of host position), it sends a detector to the host. The detection information is sent back to coordinator. The coordinator then takes action according to the result.

5.1.1 Agent model

In our protocol, a mobile agent (MA) consists of program, data, and state. Program represents the executable program to perform tasks on remote hosts. Taking the JAVA programming language as an example, the program part can be a piece of bytecode. Data represents information brought along by MA; for example, addresses of remote hosts to travel. The data part can be plain or encrypted. State represents values of variable identities of the program. We assume that the program state is serializable. Regarding to characteristics of MA in our protocol, MA has the mobility characteristic to migrate from one host to another host. MA is autonomous during the

performing of tasks. It does not need to communicate with the home host until it finishes its tasks.

5.1.2 Execution model

We assume that the program state is serializable. When MA is to migrate to a remote host, it can serialize its program state first. After migration, MA can continue its previous execution by loading its program state. Program states can be serialized into storable data.

We assume that execution of a program statement is deterministic. This means that an execution with the same program statement, program state and input must generates the same resulting program state. We assume that a remote host executes program statements sequentially.

5.1.3 System model

In our protocol, there are home host and remote hosts on the network. The hosts are totally connected. MA can migrate from one host to any other hosts on the network without routing to other hosts first. Foreign hosts can join the network anytime. Idle hosts in the network can disconnect from the network anytime.

5.1.4 Failure model

In our protocol, we only assume processor failure of remote hosts. A remote host may crash permanently or temporarily. We assume that the home host will not be permanently crashed.

5.2 The tamper-detection protocol

Here we present our protocol which keeps checking the execution trace of a mobile agent by using cooperating agents (coordinator and detector).

The terms used in the protocol are defined as follows:

- the mobile agent (MA): The agent which is sent by a home host to perform tasks. It is the target to protect.
- coordinator: An agent which coordinates the MA and detector.
- detector: An agent which detects tampering by checking the execution trace.

- home host: The host which starts the MA.
- honest host: A host which is determined by the coordinator to have not tampered with the MA and the detector. The home host is an honest host. Trusted hosts are also considered as honest hosts.
- $state_i$: The program state of the mobile agent on $host_i$ after execution.
- $input_i$: Inputs to the agent executing on $host_i$.
- obfuscated code: The code transformed from MA's code by obfuscator.
- $state'_i$: The program state of the obfuscated code on $host_i$ after execution.

The protocol requires two assumptions:

Assumption 1: The hosts allow agents to communicate with other agents on different hosts.

Assumption 2: The hosts allow agents to store data on the hosts

Assumption 3: An honest host does not attack the MA.

Assumption 1 is required because the mobile agent has to communicate with cooperating agents. Assumption 2 is required because the mobile agent has to store the execution trace on hosts. Assumption 3 is to assume that there is no collusion between honest hosts and undetermined hosts. However, Chapter 8 will describe a method to relax this assumption.

The protocol is presented below with the assumption that a mobile agent travels to $host_i$ before $host_{i+1}$.

For the home host:

1.1 Generate two sets of public-private keys

1.2 Assign a set of keys to the mobile agent

1.3 Assign the other set of keys to the coordinator

1.4 Give the public key of the mobile agent to the coordinator

1.5 Give the public key of the coordinator to the mobile agent

1.6 Compute $state_1$

1.7 Transfer the mobile agent to the next host

1.8 Give $state_1$ to the coordinator

1.9 Generate obfuscated code

1.10 Coordinator stores the obfuscated code

1.11 Coordinator records state identity mapping in obfuscated code to states in original code

The home host initializes the encrypt/decrypt element for communication between the mobile agent and coordinator. By using two pairs of public-private keys, they can send/receive messages to/from each other securely. The reason to use public-private keys instead of secret keys is that the execution trace stored by the MA should only be read by detector only. If secret keys are used, a malicious host may extract the secret keys from the MA to read the execution trace. We will not show the encrypt/decrypt steps of communication in later parts of the protocol. The home host then computes the initial state of the mobile agent. The state is given to coordinator for later detection process.

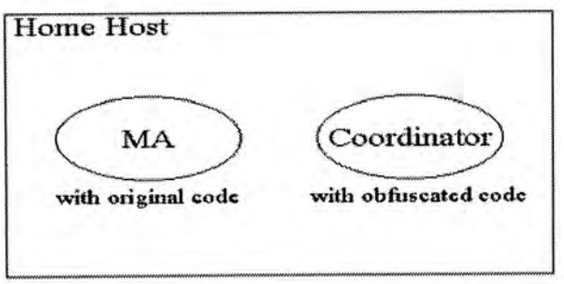

Figure 5.2: MA and coordinator in home host.

Obfuscated code is used in the re-execution process by detector (refer to Figure 5.1 round 3). The obfuscated code is generated at the home host and stored in the coordinator (Figure 5.2). In traditional execution tracing approach like [30] [11], the original code is used for re-execution process. However, in our protocol, since the platform to perform the process can be a malicious host which may try to attack the re-execution process if it has already tampered with the executed MA. If the code for re-execution is the original code, the malicious host can easily tamper with the original code and the code for re-execution twice in the same way. And the result states of re-execution will match that of original execution even though they have been tampered. Therefore, to solve the problem in our protocol, obfuscated code is used in the re-execution process instead of the original code. If a

malicious host has tampered with the states in the original MA and tries to tamper with the states again during the re-execution process, the tampered result states of the original MA and that of the obfuscated code should be different. With different result states, the protocol can detect tampering.

There are some requirements that the obfuscating transformation must satisfy in the protocol. First, since the obfuscated code is to confuse remote hosts, obfuscating transformations with high resilience (degree of resisting deobfuscating) are preferred. Preventive transforms [6] with high resilience are designed to increase the deobfuscation cost. Second, since the obfuscated code is used in re-execution, the obfuscated code must be executable. Third, the number of variables in the obfuscated code should be more than that in the original code. Most obfuscating transformations satisfy the requirement.

In addition, during compilation, a mapping of "identities of variables in executable of original code" and "identities of variables in obfuscated code" is needed to be generated and be stored in coordinator. There should be some identities without mapping to original variables since code obfuscation produces additional variables. Figure 5.3 shows an example of "identity mapping" table.

Identities of variables in executable of original code	Identities of variables in executable of obfuscated code
price_current	ali00013
	ali05150
price_best	ali82246
price_afford	ali06233
⋮	⋮
	ali00042
visit_current	ali05022
visit_best	ali00004

Figure 5.3: An example of "identity mapping" table.

Now we assume that the coordinator is on $host_i$ and the mobile agent is on $host_{i+1}$, where $host_i$ is an honest host.

For the mobile agent at $host_{i+1}$:

2.1 Perform task

2.2 Trace $input_{i+1}$

2.3 Compute $state_{i+1}$

2.4 Add $input_{i+1}$ and $state_{i+1}$ to a message

2.5 Send $state_{i+1}$ to coordinator

2.6 Encrypt the message with public key of coordinator

2.7 Store the message on $host_{i+1}$

2.8 Send message of (address of next host to migrate) to coordinator

2.9 Migrate to $host_{i+2}$

For the mobile agent at $host_{i+2}$:

3.1 Send message of (arrival to $host_{i+2}$) to coordinator

Steps 2.1 to 2.8 and 3.1 show the protocol for the mobile agent. The mobile agent performs its task on $host_{i+1}$. At the same time, it traces the inputs to its execution on the host. The set of traced inputs is represented by $input_{i+1}$. After execution with $state_i$ and $input_{i+1}$, the state of the mobile agent becomes $state_{i+1}$. Figure 5.4 illustrates an execution on $host_{i+1}$.

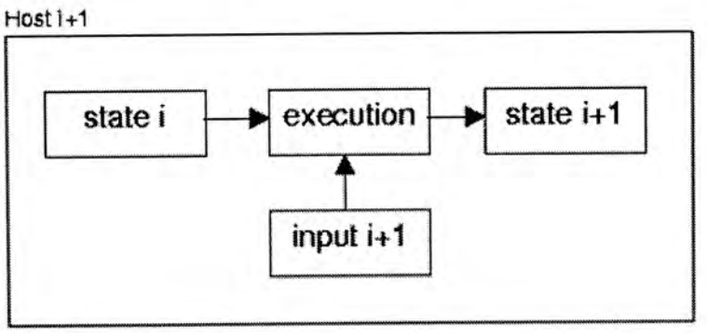

Figure 5.4: An execution on $host_{i+1}$.

The mobile agent computes $state_{i+1}$, and add $input_{i+1}$ and $state_{i+1}$ to a message (trace). $state_{i+1}$ is sent to coordinator for tamper-detection. The mobile agent then encrypts the message and stores it on $host_{i+1}$ (with Assumption 2). Since the message is encrypted with the public key of the coordinator, only coordinator and detector (with private key of coordinator)

are able to decrypt and read the message. The mobile agent sends a message about the next host to migrate to the coordinator before migration.

The mobile agent finally migrates to $host_{i+2}$. Once the mobile agent has successfully migrated to $host_{i+2}$, it sends a message to the coordinator about its arrival to a new residing host address.

Here is a consideration for host failure. If $host_{i+1}$ or $host_{i+2}$ halts when the mobile agent is residing on it, the mobile agent cannot send a message to the coordinator. A malicious host may also deny sending a message to the coordinator for the mobile agent. To overcome this possible problem, the mobile agent can send "heart-beat" messages to the coordinator after every fixed period. By receiving the "heart-beat" message, the coordinator can know that the mobile agent is still alive. If the coordinator fails to receive a "heart-beat" message from the mobile agent for some fixed period, it can assume that the mobile agent has been attacked. The host on which the mobile agent resided at last is suspected to be a malicious host.

For coordinator at $host_i$:

4.1 Receive message of (arrival at $host_{i+2}$) from the mobile agent

4.2 Generate detector by combining detector code and obfuscated code

4.3 Load $state_i$ to obfuscated code in detector

4.4 Transfer detector to $host_{i+1}$

The coordinator on $host_i$ waits for the (arrival at $host_{i+2}$) message from the mobile agent. Once the coordinator receives the message, it generates a detector by combining detector code and obfuscated code. The detector code is the part for controls such as to communicate with coordinator and to launch the obfuscated code for re-execution; while the obfuscated code is the part for execution checking. Detector loads $state_i$ to the obfuscated code according to the identity mapping table record in the coordinator. There are some identities in the obfuscated code which have no corresponding values in $state_i$. It then migrates to $host_{i+1}$ to check $input_{i+1}$ and $state_{i+1}$ stored by the mobile agent.

For detector at $host_{i+1}$:

5.1 Decrypt the message stored by the mobile agent

5.2 Execute the obfuscated code to compute $state'_{i+1}$ using $input_{i+1}$

5.3 Send $state'_{i+1}$ to coordinator

When detector has migrated to $host_{i+1}$, it decrypts the message stored by the mobile agent using the private key of the coordinator. Through the message, the detector gets $input_{i+1}$ and $state_{i+1}$. Then the detector performs the checking process. The detection mechanism is based on the execution trace approach [30]. But the obfuscated code is executed instead of the original one. This is to prevent a malicious host from tampering a piece of code in the same way twice. After the execution, the detector computes $state'_{i+1}$. $state'_{i+1}$ is sent back to the coordinator for checking.

Similar to the mobile agent, the detector can send "heart-beat" messages to the coordinator every fixed period to show that it is still alive.

Figure 5.5 shows an overview of the protocol from Step 2 to Step 5.

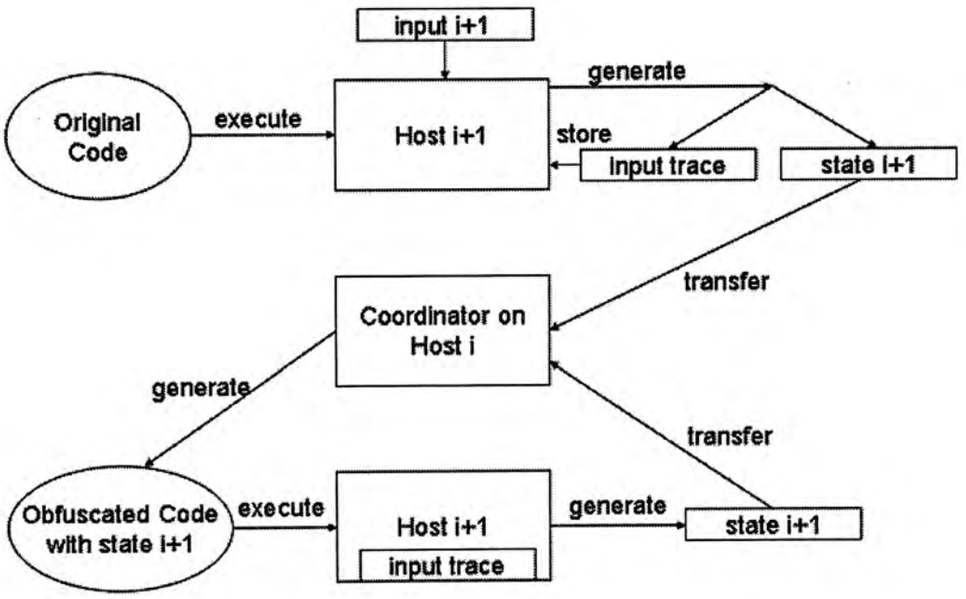

Figure 5.5: An overview of the protocol (Step 2 to Step 5).

For coordinator at $host_i$:

6.1 Receive $state'_{i+1}$ from detector

6.2 Check if $state'_{i+1}$ is valid with respect to $state_{i+1}$

6.3 If result is ok (no tampering is detected),

- send message to detector to inform it to terminate
- send message of coordinator's (address of next host to migrate to) to the mobile agent

– migrate to $host_{i+1}$

6.4 If result is not ok (tampering is detected),

– send alert message to home host

The coordinator at $host_i$ waits for $state'_{i+1}$ from the detector. It checks $state'_{i+1}$ and see if the values match the values in $state_{i+1}$. The coordinator compares the values in a pair according to the identity mapping table. Figure 5.6 shows how the checking table is generated. The values of a pair are matched if the value of an identity in $state_{i+1}$ is the same as the value if its corresponding identity in $state'_{i+1}$. If all pairs are matched, the checking result is ok (no tampering is detected). The detection mechanism is described in Chapter 4.

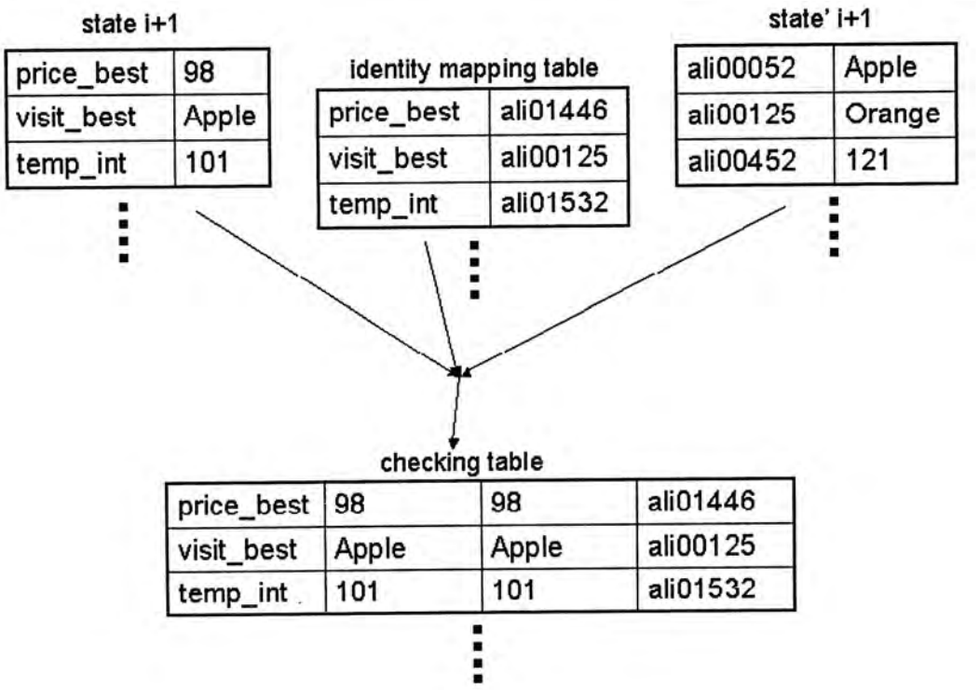

Figure 5.6: An example of generation of checking table.

If the result is ok, meaning that the host is honest, the coordinator first sends a message to the detector to inform it to terminate. It then sends a message to the mobile agent at $host_{i+2}$ stating the address of the next host the coordinator will migrate to. And then the coordinator migrates to $host_{i+1}$. Before migration, coordinator discards $state'_i$ but retains $state'_{i+1}$. The mobile agent (MA) starts to follow the protocol from Step 2.1. The new coordinator resides on an honest host.

If the result is incorrect, meaning that the host is malicious, the coordinator sends an alert message to the home host. The alert message contains the address of the malicious host and the address of the host the mobile agent is residing at the moment. The home host thus knows about the malicious host. It can then take action to handle the tampered mobile agent. For example, it may call the mobile agent back home or send a message to the host at which the mobile agent is residing to prevent any harmful execution.

Based on this protocol, if the mobile agent is not tampered along its journey, all hosts in the itinerary will be checked and found to be honest.

5.3 Fault-tolerance policy

Our protocol uses heart-beating messages to prove the existence of an agent. MA sends heart-beating messages to coordinator. Coordinator sends heart-beating messages to MA. Detector sends heart-beating messages to coordinator. Therefore, coordinator knows the locations of MA and detector while MA and detector know the location of coordinator. The length of interval of the messages depends on network status. If network traffic is busy, the interval should be longer; if network traffic is not busy, the interval can be shorter. The length of interval is defined on home host. For example, if coordinator cannot receive heart-beating message from MA beyond the interval, coordinator will send request message to MA for response. If there is still no response from MA, coordinator will assume that MA has been attacked or the remote host is crashed. And coordinator will inform home host about this failure. Similarly, MA and detector will inform home host if there is failure occured on coordinator.

If either MA or coordinator or detector is crashed, home host will assume that the execution of MA has been tampered.

5.4 Costs of the protocol

The costs of the protocol is classified into three categories. The first one is the setup cost at the home host. The second one is the session cost in each session of the protocol from Steps 2.1 to 6.3 in Section 5.2. The third is execution cost of MA compared to execution without deploying the security protocol.

Setup Cost

- 2 public-private key generation

- 1 coordinator generation
- 1 code obfuscation

The home host needs to generate two pairs of public-private key for the mobile agent and the coordinator. The generation cost is directly related to the length of the keys. So it depends on how secure the home host wants the mobile agent to be.

The coordinator consists of three parts of codes: coordinator control code, e.g. coordination procedures, state checking procedures; detector code for generating detector; and obfuscated MA code as well as the identity mapping table. Since the coordinator control code and the detector code are provided by programmer in the beginning, the generation cost is constant time.

The cost of code obfuscation refers to the time to generate obfuscated MA code and its identity mapping. The cost depends on the algorithm of obfuscator as well as complexity of original code. For example, an obfuscator contains transformation to transform looping controls will execute longer if the original code consists of many looping controls. Therefore the cost cannot be directly related to size of original code. In our protocol, an identity mapping table is stored in coordinator. The cost of generating the identity mapping table is proportional to the number of variables in the original code.

The setup cost is only introduced when a mobile agent is generated. Therefore it affects little the performance of the protocol.

Session Cost

- 1 detector generation
- 1 migration of detector
- 1 re-execution
- 1 state comparison
- 1 migration of coordinator
- Transports of messages between the mobile agents, coordinator and detector

The session cost represents the cost to detect tampering in a session. A session starts from the MA finishing execution on a host, and ends when tamper-detection result is generated. The coordinator needs to generate a detector in a session. Since the generation is done by merging detector code

and obfuscated MA code, the cost should be low (constant time). The detector migrates to a suspicious host. The migration cost depends on the host environment and the size of the detector. However, the migration cost should be approximately equal to that of the mobile agent. With the execution trace approach, the detector performs re-execution on the host. This can be time-consuming. But the time taken should be less than the original execution by the mobile agent because the detector needs not to wait for inputs this time. However, since the code is obfuscated, there is execution overhead to execute obfuscated code compared to original code. This depends on the algorithm of the obfuscator. There are many obfuscating transformations with various execution overheads [6]. The protocol only requires transformation with high resilience and addition of variables. Therefore the overhead is controllable by suitable selection. The coordinator compares the state from re-execution to the state from the original execution. The cost is proportional to the number of states. The comparison cost should be low compared with the re-execution cost and migration cost.

The session cost mainly affects the performance (time) of tamper-detection. Since MA needs not to wait for detection result, the session cost does not affect the MA's performance.

The bottleneck of session cost is the re-execution cost, we will describe a method to reduce the cost by using slicing in Chapter 9.

MA execution Cost

- Generation of execution trace

The MA execution cost represents the overhead in executing the MA. The overhead is due to the generation of the execution trace. The trace consists of sequence of inputs to MA. Therefore the cost is proportional to the number of inputs.

Besides the above costs, there are costs for transports of communication messages between the mobile agents, coordinator and detector. The sizes of these messages are very small; most of them only contain addresses of hosts. These costs depend on the network status.

5.5 Discussion

The proposed protocol offers some advantages over existing agent protection approaches. First, the agent protection task of the protocol is mostly done by cooperating agents (coordinator and detector). The hosts only need to allow mobile agents to store data on them and communicate with agents on other hosts. The hosts need not provide additional service or computation

to support the protocol. This provides flexibility for agent systems to implement the proposed security protocol as an add-on to the existing underlying protocols. The agent systems basically do not need any modifications. The proposed protocol is a flexible and "soft" tamper-detection solution to mobile agent systems.

Second, execution overhead to execution of mobile agent is low. In the proposed protocol, besides normal execution, a MA only needs to generate execution traces and communicate with coordinator. Compared with [11], MA in our protocol needs not to wait for tamper-detection result before performing tasks. The influence on MA performance in our protocol is little.

Third, the protocol can target the address of a malicious host. Since the tamper-detection is performed every time the mobile agent completes execution, the protocol can target a malicious host immediately if the detection result shows that the host has tampered with the mobile agent.

Fourth, the protocol can detect "denial of execution" attack. In existing approaches, "denial of execution" is difficult to prevent because the mobile agent travels alone. In the proposed protocol, the mobile agent is monitored by a coordinator along its journey. If the mobile agent is trapped by a malicious host by "denial of execution", it cannot send a message to the coordinator. Once the coordinator finds that the mobile agent has no response, it can assume that the mobile agent has been attacked. The coordinator can then inform the home host and the home host can send another mobile agent to continue with the unfinished task.

There are some disadvantages of the protocol. First, the protocol cannot prevent tampering. Although the protocol can detect tampering when the mobile agent is still on its journey, it cannot prevent a malicious host from tampering with the mobile agent. It is possible that a tampered agent executes on other non-malicious hosts. There is a solution to overcome this problem. We can modify the protocol so that when the mobile agent migrates to $host_{i+1}$, $host_{i+1}$ will inform the coordinator to see if host i is honest. $host_{i+1}$ will not execute the mobile agent until it has been told that host i is honest. However, this is likely to imply a significant waiting time for the answer. Usually hosts have their own security mechanisms to protect them from harmful agents. Therefore, this modification may be deployed if hosts in the agent system can be easily attacked by agents.

Second, there are lots of communications between agents in the protocol. Although the sizes of the messages are small, they still increase the network traffic. Therefore the protocol is more suitable on networks with high bandwidth.

Third, since the protocol deploys execution tracing in its detection mechanism, it inherits some limits of execution tracing. Mobile agents are required

to be single threaded. If this is not the case, an extension to the tracing mechanism is required [30].

5.6 Conclusion

In this chapter, we have presented our tamper-detection protocol for mobile agents. Since the protocol needs no trusted third party and only needs little or no modification to agent environments, the protocol is suitable for open networks. The basic idea of the protocol is to use cooperating agents to continuously monitor the mobile agents and check the executions of the mobile agents along their journey. The detection mechanism is derived from execution tracing and code obfuscation techniques. Once a malicious host is detected, the cooperating agents can inform the home host that the mobile agent is likely to have been tampered.

The proposed protocol is flexible. Since the protocol requires mostly participation by cooperating agents (coordinators and detectors), it is easy for the protocol to be implemented on existing mobile agent systems. The prerequisite is that agent environments on the hosts can provide data storage for mobile agents.

The protocol can preserve the important characteristic of asynchronous execution for mobile agents, as the home hosts are not involved in sessions of the protocol unless the mobile agents are found to have been tampered. There is no waiting time for the mobile agents in the protocol. The protocol can also detect "denial of execution" attack from malicious hosts by continuous monitoring.

There are some drawbacks for the protocol. The intensive communication in the protocol increases network traffic, meaning that the protocol is more suitable to networks with high bandwidth. The protocol inherits the limit of execution tracing that mobile agents are required to be single threaded. Since cooperating agents (coordinator and detector) are used in the protocol, the protocol is more suitable to agent systems in which an agent environment can support several mobile agents.

Chapter 6

Verification of the Protocol by BAN Logic

6.1 Introduction

In this chapter, we verify the security of data flow of our tamper-detection protocol. The verification uses BAN logic [4] which has been widely used for analyzing authentication and cryptographic protocols [1]. BAN logic is used to find out security weakness (in cryptography) and unnecessary actions of a protocol by looking into prerequisite assumptions. However, a "verification" with BAN logic does not necessarily imply that no attacks on the protocol are possible [17].

To ensure the correctness of the protocol, we need to ensure the following three requirements:

1. Program state of original code after execution on $host_{i+1}$, i.e. $state_{i+1}$, sent by MA on $host_{i+1}$ to coordinator on $host_i$ is secure.

2. Program state of obfuscated code after re-execution on $host_{i+1}$, i.e. $state'_{i+1}$, sent by detector on $host_{i+1}$ to coordinator on $host_i$ is secure.

3. $host_{i+1}$ does not know the correct "identity mapping table".

The protocol requires coordinator to check $state_{i+1}$ and $state'_{i+1}$ with the use of an "identity mapping table". To perform the checking correctly, the two states should be the ones obtained from execution and re-execution on $host_{i+1}$ respectively. This means that the two states need to be secure during transmission from $host_i$ to $host_{i+1}$. Verification of their security is provided in this chapter. (If an unknown third party without correct "identity mapping table" modifies the two states during their transmissions, coordinator

is able to find out the inconsistence of the two states during checking. The coordinator will conclude that tampering is detected.)

In addition to the security of transmissions of states, we need to ensure that $host_{i+1}$ does not know the correct "identity mapping table". In the protocol, it is possible for home host and $\{host_1, host_2, ..., host_i\}$ to get the "identity mapping table" since coordinator has travelled to them. According to the assumption of the protocol, coordinator travels to only honest hosts. Since an honest host does not attack MA, it will not send the "identity mapping table" to other hosts. However, in real-life applications, there may be hosts that pretend to be honest but try to attack MA by getting the "identity mapping table" and sending it to a malicious host, e.g. $host_{i+1}$. We have developed an extension for the protocol to handle this problem in Chapter 8.

6.2 Modifications to BAN logic

To represent our tamper-detection protocol in BAN logic, we need to make several modifications to BAN logic. The reason is that the traditional BAN logic cannot handle some of the concepts in the protocol, e.g. code obfuscation. The followings are the necessary modifications:

1. Assume that p represents a piece of executable program, $\{p\}_\alpha$ represents corresponding obfuscated program with random seed α. On the other hand, assume that d represents a piece of data, $\{d\}_k$ represents corresponding encrypted data with key k.

2. Use $|\doteq$ to represent "not know". $P \,|\doteq\, X$ means that P may know the existence of X but P does not know the content of X.

The first modification is to represent the code obfuscation concept used in the protocol. In the original BAN logic, there is no logic to represents code obfuscation. To represent the obfuscation concept, we use the encryption logic in BAN logic. For the encryption logic in BAN logic, a key is used to encrypt a piece of data. For code obfuscation, a random seed is used for transforming a piece of program. Due to the similarity, we use the encryption logic to represent the obfuscation concept of the protocol.

The second modification is an add-on to the original BAN logic. The logic is to represent that a principal does not know the content of a piece of information. In the description of our protocol, we will use this logic to assume that $host_{i+1}$ does not know the content of the correct "identity mapping table".

To verify our tamper-detection protocol with BAN logic, we will first model the protocol, goals, sub-goals, and assumptions into BAN logic notations. The protocol describes message transmissions; goals describe what the protocol is to achieve; assumptions describe necessary conditions for the protocol to achieve the goals. Then we will verify the goals of the protocol based on the protocol and assumptions.

6.3 Term definitions

In this chapter, we will use the following terms for verification.

A : the mobile agent (MA) to perform tasks

C : coordinator

D : detector

$H(i)$: $host_i$

$H(i+1)$: $host_{i+1}$

$s(i)$: $state_i$

$s(i+1)$: $state_{i+1}$

$s'(i)$: $state'_i$

$s'(i+1)$: $state'_{i+1}$

$t(i)$: $trace_i$

$t(i+1)$: $trace_{i+1}$

p : original program of MA

$\{p\}_\alpha$: obfuscated program of MA

α : random seed for generating the obfuscated program; it also represents content of "identity mapping table"

k_A : public key of A

k_C : public key of C

E : execution process on a host

F : code obfuscation process

M : identity mapping process

The terms are coherent to the terms used in our protocol described in Chapter 5.

6.4 Modeling of our tamper-detection protocol

The message transmissions of the tamper-detection protocol can be summarized into the following steps:

1. MA encrypts $trace_{i+1}$ with public key of coordinator (k_C), and stores the encrypted message on $host_{i+1}$.
2. MA encrypts $state_{i+1}$ with public key of coordinator (k_C), and sends the encrypted message to coordinator.
3. Coordinator sends detector,obfuscated code, private key of coordinator (k_C^{-1}) to $host_{i+1}$.
4. Detector decrypts $trace_{i+1}$ with k_C^{-1}. Detector executes $\{p\}_\alpha$ with $trace_{i+1}$. Detector generates $state'_{i+1}$.
5. Detector encrypts $state'_{i+1}$ with public key of coordinator (k_C), and sends the encrypts message to coordinator.

It holds that:

- $E(s(i), p, t(i+1)) = s(i+1)$
  `Execution of original program` p `with` $state_{i+1}$ `and input` $trace_{i+1}$ `will generate` $state_{i+1}$.
- $F(p, \alpha) = \{p\}_\alpha$
  `Code obfuscation of program` p `with random seed` α `will generate obfuscated program` $\{p\}_\alpha$.
- $E(s'(i), \{p\}_\alpha, t(i+1)) = s'(i+1)$
  `Execution of obfuscated program` $\{p\}_\alpha$ `with` $state'_{i+1}$ `and input` $trace_{i+1}$ `will generate` $state'_{i+1}$.

- $M(s(i+1), \alpha) = s'(i+1)$
  If there is no tampering to the program states, there exists a corresponding identity in $state'_{i+1}$ for each identity in $state_{i+1}$ with same value. The mapping process requires an "identity mapping table" which is generated during the code obfuscation process.

In BAN logic notation, the protocol can be presented as:

1. $\overset{k_A}{\mapsto} A$
   Generate key for MA.

2. $\overset{k_C}{\mapsto} C$
   Generate key for coordinator.

3. $A \rightarrow H(i+1) : \{t(i+1)\}_{k_C}$
   MA encrypts $trace_{i+1}$ on $host_{i+1}$.

4. $A \rightarrow C : \{s(i+1)\}_{k_C}$
   MA encrypts $state_{i+1}$ and sends it to coordinator.

5. $C \rightarrow H(i+1) : D, \{p\}_{\alpha}, k_C^{-1}$
   Coordinator sends detector with obfuscated code to $host_{i+1}$.

6. $D \rightarrow C : s'(i+1)_{k_C}$
   Detector encrypts $state'_{i+1}$ and sends it to coordinator.

6.5 Goals

For the protocol to correctly perform tamper-detection, the protocol needs to achieve the following three goals:

- Coordinator believes that both coordinator and MA believe $state_{i+1}$.
- Coordinator believes that both coordinator and detector believe $state'_{i+1}$.
- Coordinator believes that mapping function can function correctly.

In BAN logic notation, the goals can be presented as:

1. $C \mid\equiv C \overset{s(i+1)}{\longleftrightarrow} A$
   C `believes that` C `and` A `both believe` $state_{i+1}$.

2. $C \mid\equiv C \overset{s'(i+1)}{\longleftrightarrow} D$
   C `believes that` C `and` D `both believe` $state'_{i+1}$.

3. $C \mid\equiv M(s'(i+1), \alpha) \equiv s(i+1)$
   C `believes that the mapping function` M `is valid.`

6.6 Sub-goals

The sub-goals are used for reaching the goals during verification. From the protocol and goals, we can define the following sub-goals:

- MA and coordinator both believe that encryption with the key k_A is secure.
- Coordinator and MA both believe that encryption with the key k_C is secure.
- Coordinator and detector both believe that encryption with the key k_C is secure.

In BAN logic notation, the sub-goals can be presented as:

1. $A \overset{k_A}{\longleftrightarrow} C$
   A `and` C `both believe in` k_A.

2. $C \overset{k_C}{\longleftrightarrow} A$
   C `and` A `both believe in` k_C.

3. $C \overset{k_C}{\longleftrightarrow} D$
   C `and` D `both believe in` k_C.

6.7 Assumptions

According to the assumptions of the protocol described in Chapter 5.2, an honest host does not collude with other host to attack MA. Specifically, an honest host does not send the "identity mapping table" to other host. The followings are the assumptions for the protocol:

- $host_{i+1}$ does not know the random seed α. (obfuscation take $host_{i+1}$ a very long time to break.)
- $host_i$ ($\{host_1, host_2, ..., host_i\}$ are honest) does not send the "identity mapping table" to other hosts. Thus $host_i$ does not send the random seed α to other hosts.

In BAN logic notation, the assumptions can be presented as:

1. $E(s(i), p, t(i+1)) \equiv s(i+1)$
2. $E(s'(i), \{p\}_\alpha, t(i+1)) \equiv s'(i+1)$
   `Detector can successfully execute obfuscated program` $\{p\}_\alpha$ `and send the result state` $s'(i+1)$ `to coordinator.`
3. $H(i+1) \mid\doteq \alpha$
   $H(i+1)$ `does not know` α.
4. $C \mid\equiv \alpha$
5. $H(i) \mid\equiv C \mid\equiv \alpha$
6. $H(i+1) \mid\equiv C \mid\equiv \alpha$
   `Honest hosts` $\{host1, host2, ..., host_i\}$ `knows the "identity mapping table".`

6.8 Verification

The verification is based on the message transmissions and assumptions described in this chapter. The basic idea of the verification is described as follow:

I. k_A is public key of MA.

II. k_C is public key of coordinator.

III. MA and coordinator both believe that encryption with k_A is secure $\rightarrow$ Sub-goal(1).

IV. Coordinator and MA both believe that encryption with k_C is secure $\rightarrow$ Sub-goal(2).

V. Coordinator and detector both believe that encryption with k_C is secure $\rightarrow$ Sub-goal(3).

VI. Coordinator and MA both believe that $state_{i+1}$ is secure $\rightarrow$ Goal(2).

VII. Coordinator and detector both believe that $state'_{i+1}$ is secure $\rightarrow$ Goal(3).

In BAN logic notation, the verifications can be presented as:

1. $A \mid\equiv \overset{k_A}{\mapsto} A$
   Public key of A.

2. $C \mid\equiv \overset{k_A}{\mapsto} A$
   C knows existence of k_A.

3. $A \mid\equiv C \mid\equiv \overset{k_A}{\mapsto} A$
   A tells C about k_A.

4. $C \mid\equiv D \mid\equiv \overset{k_A}{\mapsto} A$
   C tells D about k_A.

5. $C \mid\equiv \overset{k_C}{\mapsto} C$
   Public key of C.

6. $A \mid\equiv \overset{k_C}{\mapsto} C$
   A knows existence of k_C.

7. $C \mid\equiv A \mid\equiv \overset{k_C}{\mapsto} C$
   C tells A about k_C.

8. $D \mid\equiv \overset{k_C}{\mapsto} C$
   D knows existence of k_C.

9. $C \mid\equiv D \mid\equiv \overset{k_C}{\mapsto} C$
   C tells D about k_C.

10. $\dfrac{A \mid\equiv \overset{k_A}{\mapsto} A, A \mid\equiv C \mid\equiv \overset{k_A}{\mapsto} A}{A \overset{k_A}{\longleftrightarrow} C}$
    A and C both believe in k_A $\rightarrow$ Sub-goal(1).

11. $\dfrac{C|\equiv \overset{k_C}{\mapsto} C, C|\equiv A|\equiv \overset{k_C}{\mapsto} C}{C \overset{k_C}{\longleftrightarrow} A}$
C and A both believe in k_C $\rightarrow$ Sub-goal(2).

12. $\dfrac{C|\equiv \overset{k_C}{\mapsto} C, C|\equiv D|\equiv \overset{k_C}{\mapsto} C}{C \overset{k_C}{\longleftrightarrow} D}$
C and D both believe in k_C $\rightarrow$ Sub-goal(3).

13. $A \mid\equiv s(i+1)$
A generates $state_{i+1}$.

14. $A \mid\equiv \{s(i+1)\}_{k_C}$
A encrypts $state_{i+1}$ with k_C.

15. $A \mid\equiv C \leftarrow \{s(i+1)\}_{k_C}$
A sends $\{s(i+1)\}_{k_C}$ to C.

16. $\dfrac{A|\equiv\{s(i+1)\}_{k_C}, A|\equiv C\leftarrow\{s(i+1)\}_{k_C}}{A|\equiv C \lhd \{s(i+1)\}_{k_C}}$

17. $\dfrac{C\lhd\{s(i+1)\}_{k_C}, C|\equiv k_C^{-1}}{C|\equiv s(i+1)}$
Since C knows private key k_C^{-1}, C knows $state_{i+1}$.

18. $\dfrac{C|\equiv A\rightarrow C:\{s(i+1)\}_{k_C}, C \overset{k_C}{\longleftrightarrow} A}{C|\equiv A|\equiv s(i+1)}$
C knows that A knows $state_{i+1}$.

19. $\dfrac{C|\equiv s(i+1), C|\equiv A|\equiv s(i+1)}{C \overset{s(i+1)}{\longleftrightarrow} A}$
C and A both believe in $state_{i+1}$ $\rightarrow$ Goal(1).

20. $C \mid\equiv D \mid\equiv s'(i)$

21. $D \mid\equiv s'(i)$
D knows $state'_i$.

22. $D \mid\equiv \{p\}_\alpha$
D knows $\{p\}_\alpha$.

23. $C \mid\equiv D \mid\equiv k_C^{-1}$

24. $D \mid\equiv k_C^{-1}$
    D knows private key k_C^{-1}.

25. $$\frac{D|\equiv H(i+1) \lhd \{t(i+1)\}_{k_C}, D \overset{k_C^{-1}}{\longleftrightarrow} C}{D|\equiv t(i+1)}$$
    Since D knows private key k_C^{-1}, D knows $t(i+1)$.

26. $$\frac{D|\equiv E(s'(i), \{p\}_\alpha, t(i+1)) \equiv s'(i+1)}{D|\equiv s'(i+1)}$$
    After re-execution, D generates $state'_{i+1}$.

27. $D \mid\equiv \{s'(i+1)\}_{k_C}$
    D encrypts $state'_{i+1}$ with k_C.

28. $D \mid\equiv C \leftarrow \{s'(i+1)\}_{k_C}$
    D sends $\{s'(i+1)\}_{k_C}$ to C.

29. $$\frac{D|\equiv \{s'(i+1)\}_{k_C}, D|\equiv C \leftarrow \{s'(i+1)\}_{k_C}}{D|\equiv C \lhd \{s'(i+1)\}_{k_C}}$$

30. $$\frac{C \lhd \{s'(i+1)\}_{k_C}, C|\equiv k_C^{-1}}{C|\equiv s'(i+1)}$$
    Since C knows private key k_C^{-1}, C knows $state'_{i+1}$.

31. $$\frac{C|\equiv D \rightarrow C : \{s'(i+1)\}_{k_C}, C \overset{k_C}{\longleftrightarrow} D}{C|\equiv D|\equiv s'(i+1)}$$
    C knows that D knows $state'_{i+1}$.

32. $$\frac{C|\equiv s'(i+1), C|\equiv D|\equiv s'(i+1)}{C \overset{s'(i+1)}{\longleftrightarrow} D}$$
    C and D both believe in $state'_{i+1}$ → Goal(2).

33. $H(i) \mid\equiv \alpha$

34. $H(i) \mid\equiv H(i+1) \mid\doteq \alpha$
    Honest host does not tell content of "identity mapping table" to undetermined hosts.

35. $C \mid\equiv H(i) \mid\equiv H(i+1) \mid\doteq \alpha$
    C believes that honest host does not tell content of "identity mapping table" to undetermined hosts.

36. $C \mid\equiv H(i+1) \mid\doteq \alpha$
    C believes that $H(i+1)$ does not know content of "identity mapping table".

37. $\dfrac{C\mid\equiv s(i+1), C\mid\equiv s'(i+1), C\mid\equiv H(i+1)\mid\doteq\alpha}{C\mid\equiv M(s'(i+1),\alpha)\equiv s(i+1)}$
    Since C believes that $state_{i+1}$, $state'_{i+1}$, and α are secure, C believes that the mapping function M is valid $\rightarrow$ Goal(3).

6.9 Conclusion

From the verification, we can draw the following three conclusions:

1. It is secure for MA on $host_{i+1}$ to send $state_{i+1}$ to coordinator on $host_i$.
2. It is secure for detector on $host_{i+1}$ to send $state'_{i+1}$ to coordinator on $host_i$.
3. Assume that $host_{i+1}$ does not know the "identity mapping table", coordinator can correctly perform tamper-detection according to $state_{i+1}$ and $state'_{i+1}$.

The results show that the message transmission in the protocol is secure, and thus coordinator is able to perform tamper-detection by checking program states $state_{i+1}$ and $state'_{i+1}$. However, a malicious host may try to bypass tamper-detection in two ways. The first way is to deobfuscate the obfuscated program. If a malicious host can successfully deobfuscate the obfuscated program, it is able to construct the "identity mapping table". As described in Section 4.3.1, deobfuscation can be a very time-consuming process similar to breaking encryption. Therefore it is not easy for a malicious host to bypass tamper-detection in this way. The second way is to collude with an honest host. Although our protocol assumes that an honest host does not collude with undetermined hosts, this assumption may be invalid in real applications. Therefore we have developed an extension to handle this problem in Chapter 8.

Chapter 7

Experimental Results Related to the Protocol

7.1 Introduction

We have conducted experiments to evaluate the performance of our tamper-detection protocol. The experiments measure the overheads to the mobile agent under different settings. The settings include running without using the protocol, running with using protocol. From the experiments, we can observe the overheads of deployment of the protocol.

Java is popular in the development of mobile agent systems. Although it is not specifically designed for mobile agents, the inherent support of code mobility and platform independence makes Java a good choice to develop mobile agent systems [3] [7]. In the experiments, we use the IBM Aglets [14] [15] as the mobile agent platform. It is one of the most popular mobile agent platforms.

7.2 Experiment environment

We have used two sets of experiment environment. They are "single machine" environment and "two machines network" environment. The "single machine" environment uses a computer to simulate multiple hosts, therefore, there is no network traffic. On the other hand, the "two machines network" environment uses two computers to establish a network where mobile agents need to migrate from one computer to another computer. By using the two different environment settings, we can obtain experimental results with and without influence of network traffic and communications factors. The experiment environment is listed in the followings.

Single machine environment:

Machine : Intel Pentinum IV 1.4GHz with 256MB RAM

Operating System : Microsoft Windows 2000

Execution Platform : Java 1.1.8

Agent Platform : IBM Aglets 1.1.0

Two machines network:

Two Machines : Intel Pentinum II 450MHz with 128MB RAM

Operating System : Microsoft Windows 2000

Execution Platform : Java 1.1.8

Agent Platform : IBM Aglets 1.1.0

During the experiments, other programs are closed to minimize their influence on the experimental results.

7.3 Experiment procedures

In each experiment, seven Aglets servers "Tahiti" are setup on the machine(s). The servers have the same IP addresses but different port numbers of 1000, 2000, 3000, 4000, 5000, 6000, and 7000 respectively. The servers with port number 1000 and 7000 are defined as home hosts. A mobile agent is generated on home host with port number 1000. The agent will travel from home host to server of port number 2000, then to 3000 and so on. Finally the mobile agent will migrate to the server of port number 7000 and stop. Among the hosts, there are 5 remote hosts (port numbers from 2000 to 6000). In each experiment, five different numbers of remote hosts are used. For example, one remote host means to use servers of port numbers 1000, 2000 and 7000; while two remote hosts means to use servers of port numbers 1000, 2000, 3000, and 7000. The measured times are counted from initialization of the mobile agent to its arrival at home host.

Under the "single machine" environment, the Aglets servers are setup on single machine. On the other hand, under the "two machines network" environment, servers with port number 1000, 3000, 5000 and 7000 are setup on one machine while servers with port number 2000, 4000 and 6000 are setup on the other machine.

The mobile agent will be executed under two different settings. The two settings are "without using the protocol" and "with using the protocol". The first setting "without using the protocol" means that the mobile agent is executed without deployment of our tamper-detection protocol. The second setting "with using the protocol" means that the mobile agent is executed with deployment of our protocol. In this way, the overheads (in terms of time) of the protocol to the mobile agent can be measured.

Besides, there are two parameters for each settings. They are "input and "cycle". The "input" value represents the number of external variable inputs from hosts to agents. It is done by reading a 10-byte string for each input. The "cycle" value represents the number of cycles where every cycle calculates an integer summation of 1000 values. The usage of the parameters are similar to the ones used in [11].

In each experiment, five trials are performed and we take the average results for plotting the charts. The data can be referred to in Appendix A.

7.4 Experiment implementation

In the experiment, we have programmed two mobile agents. One is a plain mobile agent which only reads external inputs from remote hosts and perform the summation computations mentioned in Section 7.3. Another is the mobile agent which deploys our tamper-detection protocol. The protected agent also performs the tasks of the plain agent. The heart-beating messaging is excluded in the experiment since we want to concentrate on the execution overhead to the protected agent. The overhead of heart-beating also depends on network status very much. In the following we describe the structures of the plain agent and the protect agent. Code segments are used for explanations.

Structure of the plain agent

Step 1. Initialization of itinerary: Use a string array to store the hosts in the order of itinerary. Assume that the machines used for the experiments have the IP addresses "123.123.123.123" and "123.123.123.124" respectively.
In "single machine" environment, the itinerary is defined as:

```
String[] itinerary = ("atp://123.123.123.123:1000",
"atp://123.123.123.123:2000", "atp://123.123.123.123:3000",
"atp://123.123.123.123:4000", "atp://123.123.123.123:5000",
"atp://123.123.123.123:6000", "atp://123.123.123.123:7000");
```

In "two machines network" environment, the itinerary is defined as:

```
String[] itinerary = ("atp://123.123.123.123:1000",
```

```
"atp://123.123.123.124:2000", "atp://123.123.123.123:3000",
"atp://123.123.123.124:4000", "atp://123.123.123.123:5000",
"atp://123.123.123.124:6000", "atp://123.123.123.123:7000");
int host_order = 0;
int final_host = 6;
```

Step 2. Initialization of time: Initialize the starting time.

```
private Date initTime, endTime;
initTime = new Date();
```

Step 3. Looping control: Do Step 4 to Step 7 until the agent reaches the host "atp://123.123.123.123:7000".

```
while(host_order < final_host){...};
```

Step 4. Read "input": Read external variable inputs from the host store in file "C:\Aglets\data.txt". Each input is a 10-byte string. INPUT_NUMBER represents the number of inputs for the experiment.

```
int i;
String dummy = null;
File infile = new File("C:\Aglets\data.txt");
for (i=0; i<INPUT_NUMBER; i++)
   dummy = (String)(FileInputStream(infile).read(10));
```

Step 5. Perform "cycle" computation: Calculate an integer summation of 10000 values for each cycle. CYCLE_NUMBER represents the number of cycles for the experiment.

```
long i, j, number;
for (j=0; j<CYCLE_NUMBER; j++){
number = 0;
for (i=1; i<10000; i++)
   {number = number + i;}
}
```

Step 6. Measure time: Measure the time used from starting up to now.

```
endTime = new Date();
long usedTime = 0;
used Time = endTime.getTime() - initTime.getTime();
System.out.println(usedTime);
```

Step 7. Migration to next host.

```
host_order = host_order + 1;
```

```
URL next_host = new URL(host_order);
dispatch(next_host);
```

The plain agent is a simple agent which migrates from a host to another. It perform the "input" reading task and summation computation task on each host.

Structure of the protected agent

Step 1. Initialization of itinerary: Use a string array to store the hosts in the order of itinerary. Same as Step 1 of the plain agent.
In "single machine" environment, the itinerary is defined as:

```
String[] itinerary = ("atp://123.123.123.123:1000",
"atp://123.123.123.123:2000", "atp://123.123.123.123:3000",
"atp://123.123.123.123:4000", "atp://123.123.123.123:5000",
"atp://123.123.123.123:6000", "atp://123.123.123.123:7000");
```

In "two machines network" environment, the itinerary is defined as:

```
String[] itinerary = ("atp://123.123.123.123:1000",
"atp://123.123.123.124:2000", "atp://123.123.123.123:3000",
"atp://123.123.123.124:4000", "atp://123.123.123.123:5000",
"atp://123.123.123.124:6000", "atp://123.123.123.123:7000");
int host_order = 0;
int final_host = 6;
```

Step 2. Initialization of time: Initialize the starting time. Same as Step 2 of the plain agent.

```
private Date initTime, endTime;
initTime = new Date();
```

Step 3. Generate coordinator: Create the coordinator agent from code base. The code base is the bytecode generated from source code of the coordinator. The package for the coordinator agent is "experiment.CoorAglet".

```
try{
   getAgletContext().createAglet(
    getCodeBase(),"experiment.CoorAglet",nullObject);
}
catch(Exception exp){
   ...}
```

Step 4. Generate public/private key pairs and the obfuscated code: Generate two pairs for public/private keys. One for the protected agent and one for the coordinator. Besides, generate an obfuscated code from the protected agent's code.

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "SUN");
KeyPair pair = keyGen.generateKeyPair();
PrivateKey pri_key = pair.getPrivate();
PublicKey pub_key = pair.getPublic();
```

In the experiments, the code obfuscation is done by pre-defined substitution of statements.
We use "loop blocking" transformation to obfuscate the statements:

```
for (i=1; i<10000; i++)
   number = number + i;
```

into:

```
long k;
for (k=1; k <=10000; k+=10000)
for (i=1; i< min(10000,k+9999); i++)
   number = number + i;
```

Step 5. Looping control: Do Step 6 to Step 13 until the agent reaches the host "atp://123.123.123.123:7000". Similar to Step 3 of the plain agent.

```
while(host_order < final_host){...};
```

Step 6. Send message to coordinator about current host: Send a message with address of current host to the coordinator "experiment.CoorAglet". The protected agent only sends the message and does not wait for reply.

```
AgletProxy proxy = get AgletContext().createAglet(
   getCodeBase(),"experiment.CoorAglet",null);
try{
   proxy.sendMessage(new Message("arrival",itinerary[host_order]);
}
catch(MessageException exp){
   ...}
```

Step 7. Read "input": Read external variable inputs from the host store in file "C:\Aglets\data.txt". Each input is a 10-byte string. INPUT_NUMBER represents the number of inputs for the experiment. Same as Step 4 of the plain agent.

```
int i;
String dummy = null;
File infile = new File("C:\Aglets\data.txt");
for (i=0; i<INPUT_NUMBER; i++)
   dummy = (String)(FileInputStream(infile).read(10));
```

Step 8. Trace execution: This step works together with previous step (Step 6). The protected agent only trace the external variable inputs from the host. It is done by writing the values of the external variable inputs to file "C:\Aglets\trace.txt" stoerd in the host. Therefore, by combining Step 7 and 8, the program statements in Step 6 are modified into the following statements.

```
int i;
String dummy = null;
File infile = new File("C:\Aglets\data.txt");
File outfile = new File("C:\Aglets\trace.txt");
for (i=0; i<INPUT_NUMBER; i++)
   dummy = (String)(FileInputStream(infile).read(10));
   FileOutputStream(outfile).write(dummy);
```

Step 9. Encrypt the execution trace: Encrypt the trace with the public key of coordinator.

Step 10. Perform "cycle" computation: Calculate an integer summation of 10000 values for each cycle. CYCLE_NUMBER represents the number of cycles for the experiment. Same as Step 5 of the plain agent.

```
long i, j, number;
for (j=0; j<CYCLE_NUMBER; j++){
number = 0;
for (i=1; i<10000; i++)
   {number = number + i;}
}
```

Step 11. Send message to coordinator about next host to migrate: Send a message with address of next host to the coordinator "experiment.CoorAglet". The protected agent only sends the message and does not wait for reply.

```
AgletProxy proxy = get AgletContext().createAglet(
   getCodeBase(),"experiment.CoorAglet",null);
try{
   proxy.sendMessage(new Message("migration",itinerary[host_order+1]);
}
catch(MessageException exp){
   ...}
```

Step 12. Measure time: Measure the time used from starting up to now. Same as Step 6 of the plain agent.

```
endTime = new Date();
long usedTime = 0;
```

```
used Time = endTime.getTime() - initTime.getTime();
System.out.println(usedTime);
```

Step 13. Migration to next host. Same as Step 7 of the plain agent.

```
host_order = host_order + 1;
URL next_host = new URL(host_order);
dispatch(next_host);
```

The protected agent can be considered as a plain agent with using our tamper-detection protocol. The additional parts of the protected agent include generation of coordinator, generation of public-private key pairs, code obfuscation, and communications with the coordinator. Please note that the obfuscating transformation used in the experiments are constant-time complexity. But in real case, the complexity of an obfuscating transformation usually depends on number of control statements in the code.

7.5 Experimental results

By assigning parameter "input" to either 1 or 100 and assigning parameter "cycle" to either 1 or 10000, we have generated the experimental results shown in Figures 7.1, 7.2, 7.3, and 7.4 under "single machine environment" setting.

Figure 7.1 shows the results under the configuration that there is almost no input and computation for the mobile agent. In Figure 7.2, there are some inputs for the mobile agent but almost no computation. In Figure 7.3, there are heavy computations for the mobile agent but almost no input. In Figure 7.4, there are both heavy computations and some inputs for the mobile agent.

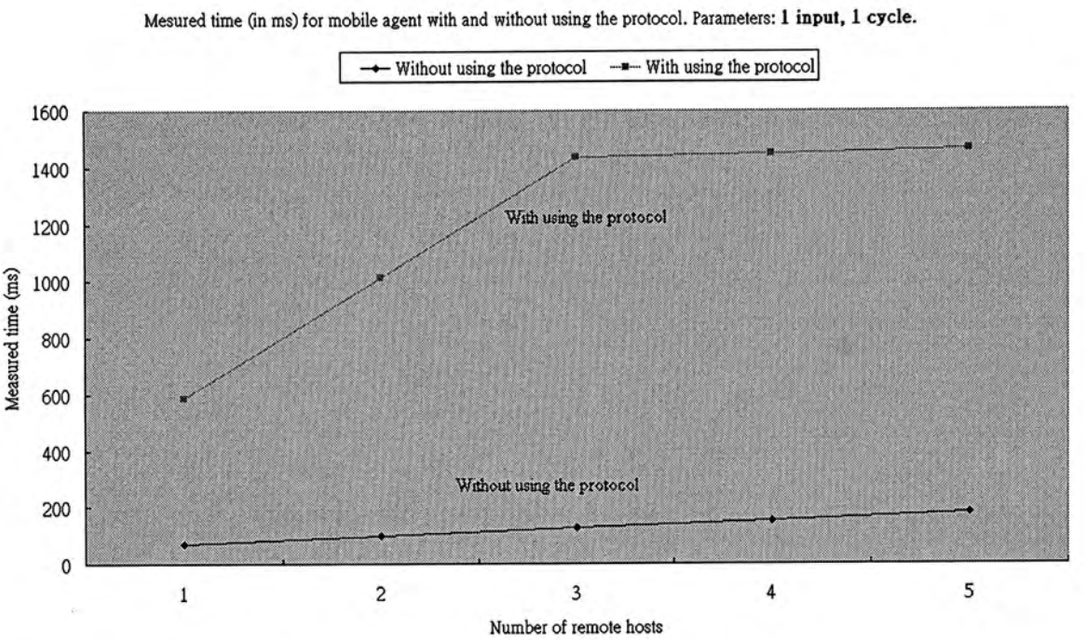

Figure 7.1: Measured time for mobile agent with and without using the protocol. Parameters: **1 input, 1 cycle**. Single machine environment.

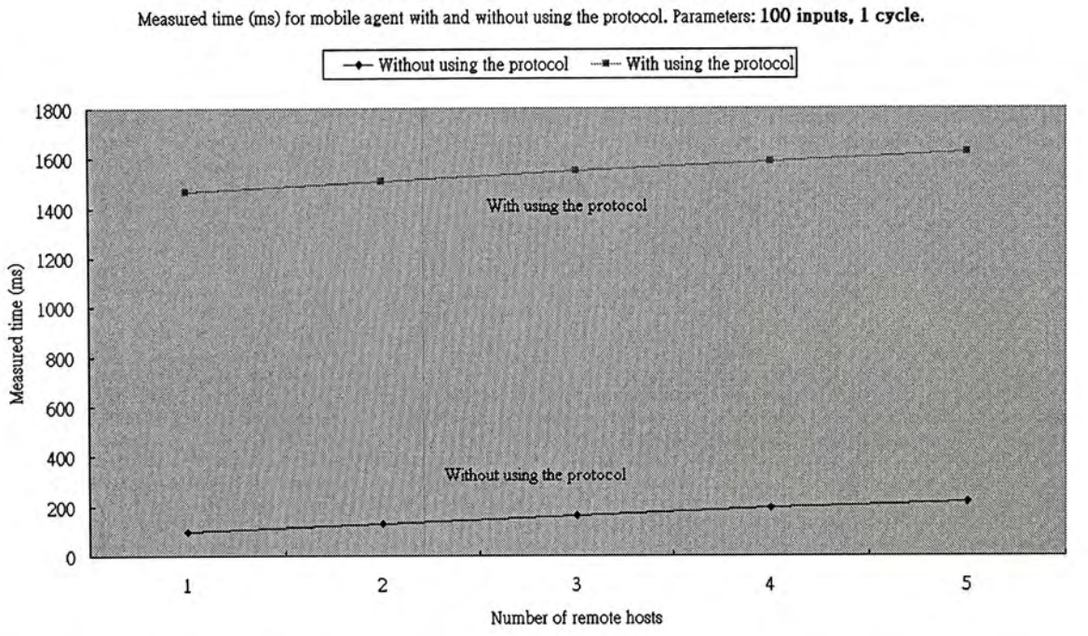

Figure 7.2: Measured time for mobile agent with and without using the protocol. Parameters: **100 inputs, 1 cycle**. Single machine environment.

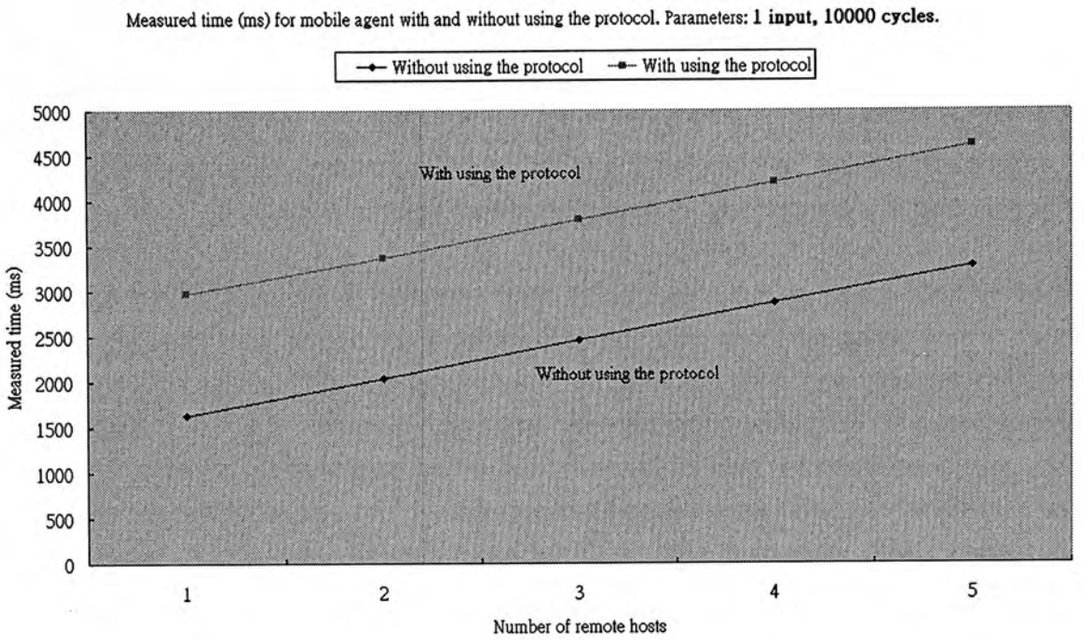

Figure 7.3: Measured time for mobile agent with and without using the protocol. Parameters: **1 input, 10000 cycles**. Single machine environment.

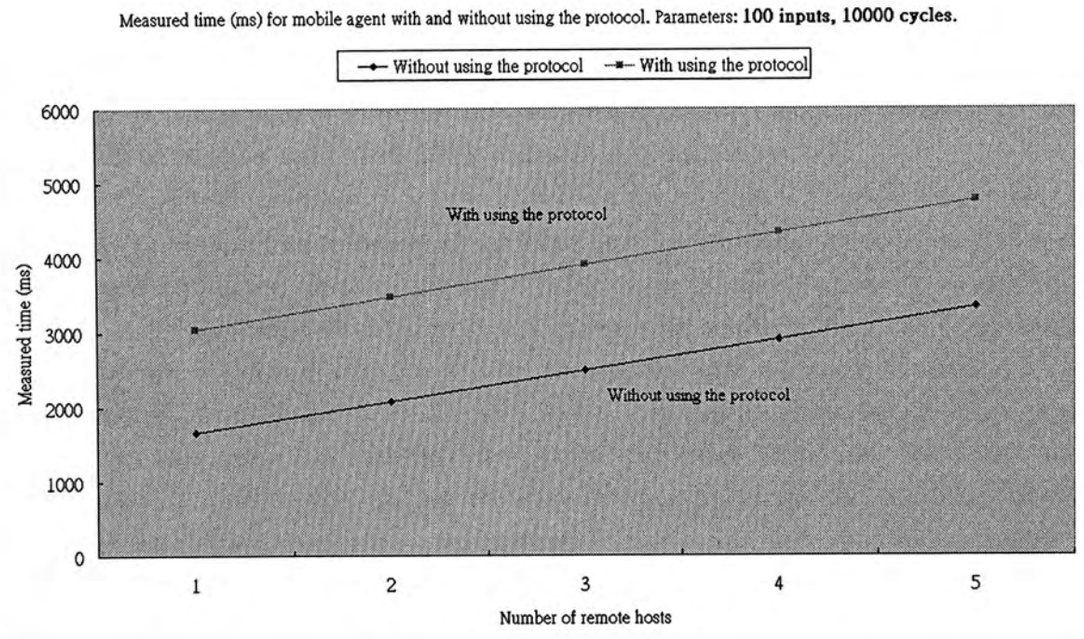

Figure 7.4: Measured time for mobile agent with and without using the protocol. Parameters: **100 inputs, 10000 cycles**. Single machine environment.

¿From Figure 7.1 to 7.4, we observe that there is a significant one-time overhead to deploy our protocol. The offsets between graphs of "using" and "without using" the protocol imply this initialization overhead. The cost is mainly contributed by generation of coordinator in the beginning at home host. Since it is only consumed once in the itinerary of the mobile agent, it does not affect the agent's performance much if the agent's itinerary is not too short.

On the other hand, we observe that the runtime overhead for the mobile agent to deploy our protocol is very little. We can see that the slopes of the two graphs in each figure are close. This is especially obvious in Figures 7.2, 7.3 and 7.4. The slopes measure the average amount of time used on each remote host. The time includes migration and execution time. Since the slopes are the graphs of "using the protocol" and that of "without using the protocol" are close, we can deduce that the runtime overheads of the protocol to the mobile agent are very smaller.

Parameters	Without the protocol	With the protocol	Ratio
1 input,1 cycle	27.6	220.35	7.98
100 inputs,1 cycle	30.05	39	1.30
1 input,10000 cycles	414.65	411.15	0.99
100 inputs,10000 cycles	417.1	433.1	1.04

Table 7.1: Average time used for the mobile agent to execute on an additional remote host. Single machine environment experiment.

Table 7.1 shows the average time used for the mobile agent if an additional remote host is added in its itinerary. The time is calculated by dividing the increase in time measured from supporting one remote host to four remote hosts by four. Taking Figure 7.2 as example, the average time for "using the protocol" equals to $(218.4 - 98.2)/4 = 30.05$. The ratio field equals to the average time for "using the protocol" over that for "without using the protocol". ¿From the table, we can investigate how external inputs and computations affect runtime overheads.

In the table, the second to fourth entries (rows) provide more reasonable results than the first entry. From the results of the second entry (100 inputs,1 cycle) and the third entry (1 input,10000 cycles), we can see that the 100 inputs affect our protocol to generate an 30 percent runtime overhead. On the other hand, our protocol is not quite affected by the 10000 cycles. This shows that the overhead to pure computation is negligible. In addi-

tion, by comparing the results of the third entry (1 input,10000 cycles) and the fourth entry (100 inputs,10000 cycles), we can see that the 100 inputs do affect the protocol. This is because our protocol needs to trace external variable inputs for tamper-detection. The action to write information to the trace contributes to the overhead.

Figure 7.1 to 7.4 shows the experimental results under "two machines network" environment. The results also show that the number of inputs affect the performance of the mobile agent with using the protocol in larger extent than the number of execution cycles. In addition, the communication overheads of the protocol are more significant in the "two machines network" environment. As a result, from Table 7.2, the overall performance overheads of the protocol are higher under the environment setting.

Parameters	Without the protocol	With the protocol	Ratio
1 input,1 cycle	42.45	133.75	3.15
100 inputs,1 cycle	55.05	486.35	8.83
1 input,10000 cycles	1166.2	1884.2	1.62
100 inputs,10000 cycles	1295.45	2067.9	1.60

Table 7.2: Average time used for the mobile agent to execute on an additional remote host. Two machines network environment experiment.

7.6 Conclusion

We have conducted experiments to evaluate the performance of our protocol. From the experimental results, we can see that the runtime overheads for deploying the protocol are small. Execution tracing of external inputs to the mobile agent contributes most to the runtime overhead. Therefore the influence of external variable input to mobile agent using our protocol is greater than that of computation size.

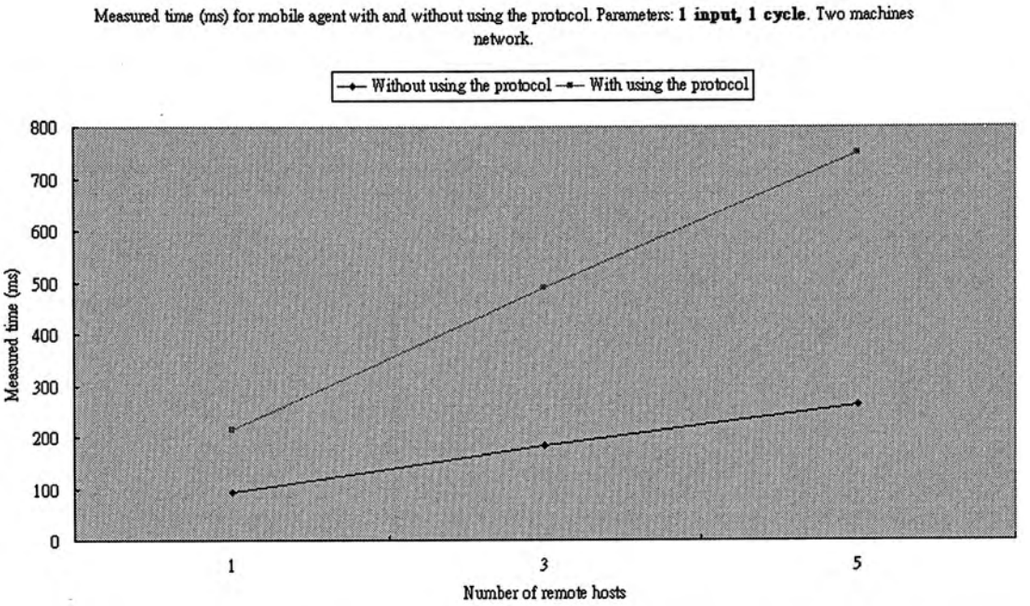

Figure 7.5: Measured time for mobile agent with and without using the protocol. Parameters: **1 input, 1 cycle**. Two machines network.

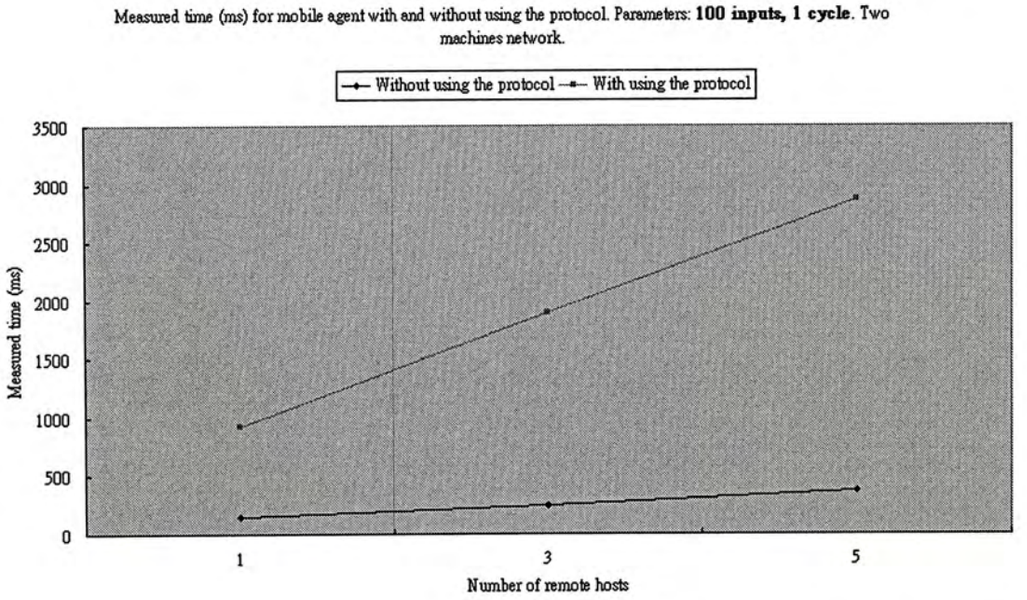

Figure 7.6: Measured time for mobile agent with and without using the protocol. Parameters: **100 inputs, 1 cycle**. Two machines network.

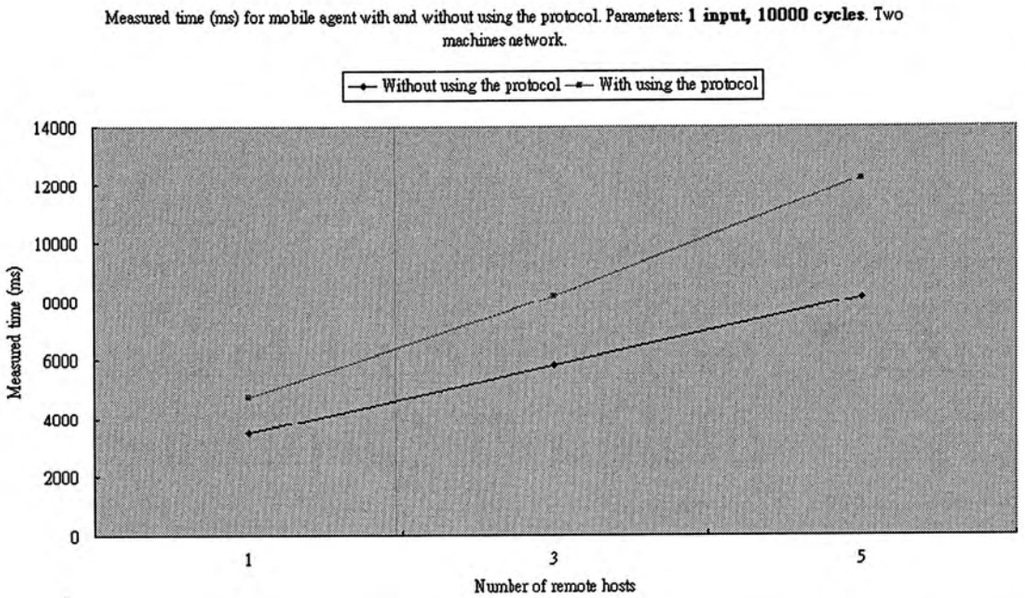

Figure 7.7: Measured time for mobile agent with and without using the protocol. Parameters: **1 input, 10000 cycles**. Two machines network.

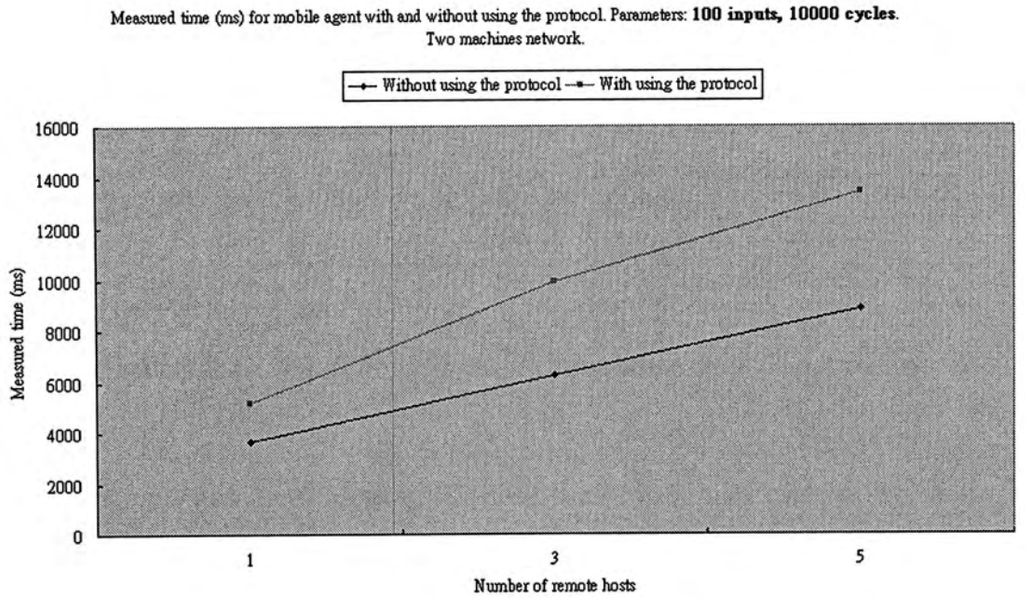

Figure 7.8: Measured time for mobile agent with and without using the protocol. Parameters: **100 inputs, 10000 cycles**. Two machines network.

Chapter 8

Extension to Solve the "Fake Honest Host" Problem

8.1 Introduction

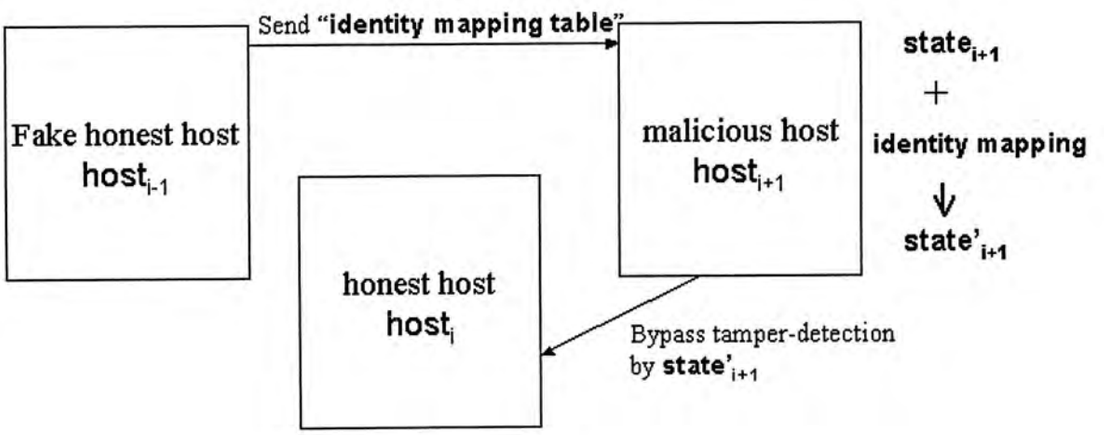

Figure 8.1: An example of "fake honest host" problem.

The protocol assumes that an honest host does not collude with undetermined host(s). In real-life applications, it is not easy to verify this assumption. Some hosts may pretend to be honest at the beginning, i.e. do not attack MA during MA's execution; but later collude with malicious hosts to attack the MA. We call this problem the "fake honest host" problem. Figure 8.1 shows an example of the problem. One way to attack MA is to spy on the content of the "identity mapping table" and send it to malicious host(s). If a malicious host gets the "identity mapping table", it knows the mapping

between identities of MA's original code to that of obfuscated code. According to the mapping, the malicious host (assuming it is $host_{i+1}$) can generate a valid $state'_{i+1}$ from $state_{i+1}$.

In this chapter, we introduce an extension to the original tamper-detection protocol to solve the problem. Assuming that in the current moment, MA is executing on $host_{i+1}$ and coordinator is ready on $host_i$. $\{host_1, host_2, ..., host_i\}$ are honest hosts while $host_{i+1}$ is undetermined. Assume that $host_{i+1}$ is malicious and wants to bypass the tamper-detection by colluding with an honest host, the extension enables the tamper-detection protocol to detect collusion between $host_{i+1}$ and $\{host_1, host_2, ..., host_{i-1}\}$. This can greatly relax the assumption that honest hosts and undetermined host do not collude. However, the extension needs the intervention of the home host and it still cannot handle the collusion between $host_{i+1}$ and $host_i$.

8.2 The method to solve the "fake honest host" problem

8.2.1 Basic idea

The basic idea of the solution is to modify the "identity mapping table" when coordinator is migrated to an honest host. Home host randomly generates the information of modification and sends it to coordinator. As a result, the mapping tables of the honest hosts are different. If a malicious $host_{i+1}$ colludes with $\{host_1, host_2, ..., host_{i-1}\}$ to get an "identity mapping table", the table is not the same with the table for coordinator on $host_i$. Therefore coordinator on $host_i$ can detect tampering if $host_{i+1}$ tries to use the table to generate $state'_{i+1}$.

8.2.2 Description of the method

First, on home host, we need to insert variable *track_var* to the obfuscated program so that *track_var* points to a variable in the state of the original MA program. For example:

```
int track_var;
track_var = (int)(best_price);
```

Correspondingly, there is a field map_{track} in "identity mapping table" representing the mapping between identity of *track_var* and identity of *best_price*. The modified obfuscated program and "identity mapping table" are stored

in coordinator in the same way as in the original protocol. Figure 8.2 shows the initialization of track variable on home host.

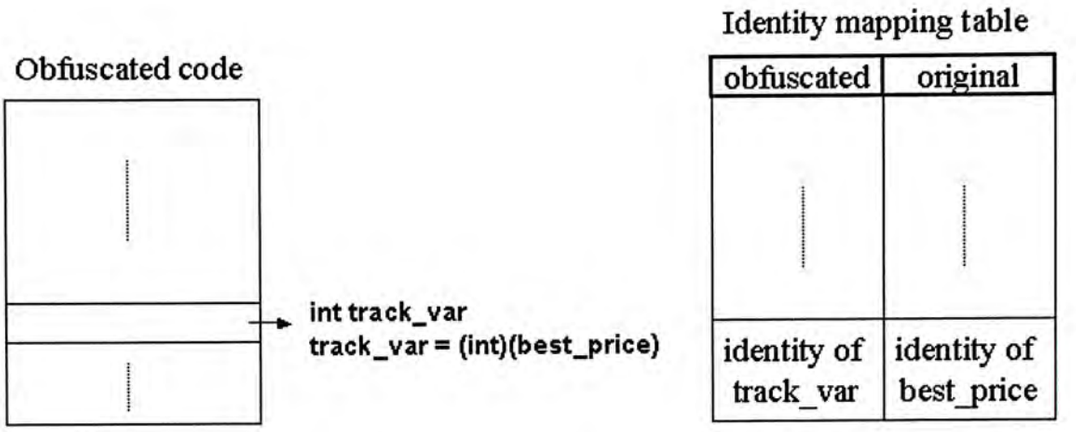

Figure 8.2: Initialization of track variable on home host.

Afterwards, assume that coordinator finishes checking $state_3$ and $state'_3$ on $host_2$ and result is okay, coordinator migrates to $host_3$. After migration, coordinator sends request of *track_var* information to home host. Home host will randomly select a variable in the state of the original MA program given that the variable has not been selected for mapping with *track_var* before. Home host then encrypts the variable information and sends it to coordinator. After receiving the variable information, coordinator modifies the obfuscated code so that *track_var* points to the newly selected variable. Accordingly, coordinator updates the field map_{track} in "identity mapping table". When MA on $host_4$ finishes execution, coordinator on $host_2$ generates detector using the modified obfuscated code. Since obfuscated code is modified, the detector generated on $host_3$ is different to that on $host_2$. The detector and coordinator then perform tamper-detection as described in the original protocol in Chapter 5. Figure 8.3 shows the steps of the extension to handle the "fake honest host" problem.

Assuming that $host_4$ is malicious and colludes with $host_2$, $host_2$ is a "fake honest host". The "fake honest host" $host_2$ records the "identity mapping table" $table_2$ and sends it to malicious host $host_4$. $host_4$ tries to bypass tamper-detection by generating $state'_4$ from $state_4$ and $table_2$. However, a correct $state'_4$ can only be generated from $state_4$ and $table_3$. As a result, when coordinator checks the incorrect $state'_4$ with $table_3$, coordinator can find that $host_4$ is likely to collude with an honest host since only the mapping pair map_{track} is not coherent. Figure 8.4 shows that the collusion between $host_4$

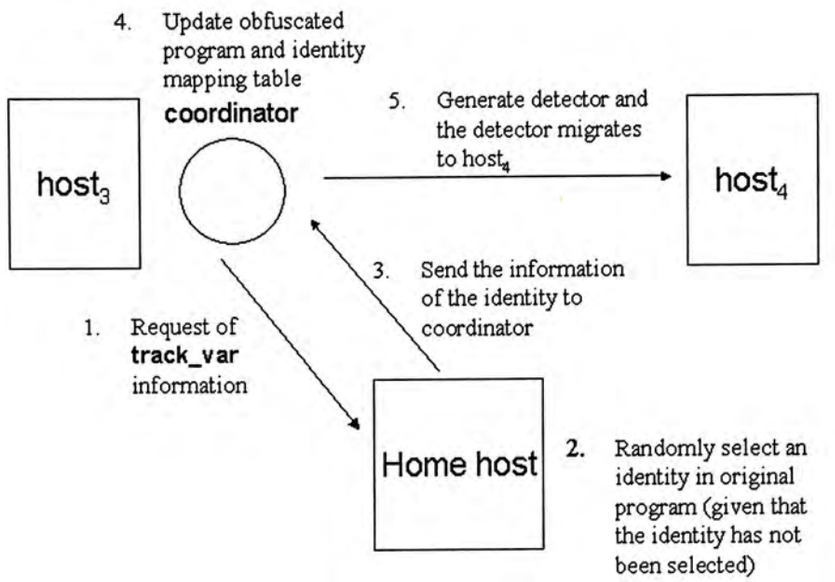

Figure 8.3: Steps of method to handle "fake honest host" problem.

and $host_2$ can be detected.

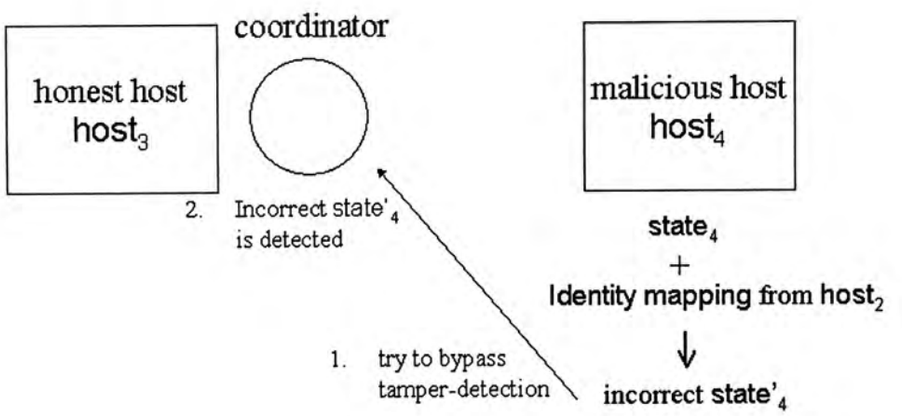

Figure 8.4: Detection of collusion.

8.3 Conclusion

In our tamper-detection protocol, we assume that honest hosts do not collude with malicious host. In this chapter, we described an extension to the protocol to relax the assumption. The method is to modify the "identity

mapping table" of coordinator on each honest host. Since the modification information is randomly generated by home host, a host cannot deduce the exact "identity mapping table" of coordinator on other host. The mapping tables are different on the honest hosts. Using the method, even if a malicious host $host_{i+1}$ colludes with "fake honest host" $\{host_{i-1}, host_{i-2}, ...\}$, coordinator on $host_i$ can still correctly detect tampering.

Chapter 9

Performance Improvement by Program Slicing

9.1 Introduction

In this chapter, we present an extension to our protocol to shorten the detection time by trading off security level. It may be odd to lower security level, but there can be situation where MA has pre-defined trustworthy level to remote hosts. For example, assume that MA and a remote host are owned by the same owner, the MA may conclude that a lower security level is acceptable if detection time can be shortened. A shorter detection time implies that the remote host can save the time to do other jobs. We deploy program slicing techniques to achieve this task.

9.2 Deployment of program slicing

A program slice consists of the parts of a program that (potentially) affect the values computed at some point of interest, referred to as a slicing criterion. The parts of a program which have a direct or indirect effect on the values computed at a slicing criterion C are called the program slice with respect to criterion C. The task of computing program slices is called program slicing. Static slicing only uses statically available information (e.g. data flow and control flow dependences) for computing slices [28].

In our protocol, the time taken by the detection process in our protocol is approximately equal to that of MA's execution. This is because the code needs to be executed completely to generate result program states. The protocol can use program slicing techniques to shorten the runtime overhead of detection. The idea is to generate two slices, where variables of one slice are

more important (or likely to be attacked) and that of the second slice are less important. Regarding to the tamper-detection mechanism of the protocol, a complete detection needs to execute the whole piece of code to generate the result states for checking. By slicing the code into two, MA's owner can select incomplete detection to trade for performance. An incomplete detection will only execute the more important slice and check its result states, while a complete detection needs to execute the less important slice too. The incomplete detection provides security level in between full detection and no detection. The decision of performing incomplete or complete detection is made by MA's owner. The security level can depend on remote host's reputation. For example, MA may need lower security level on a familiar host.

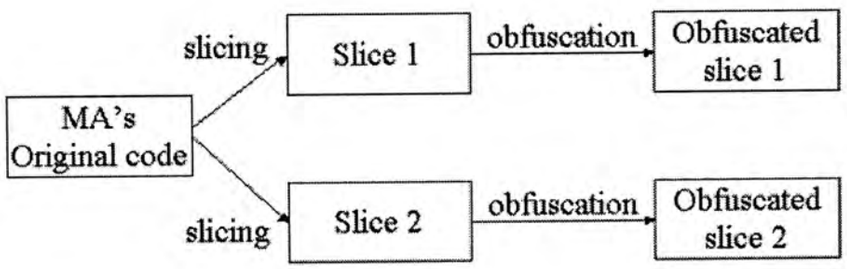

Figure 9.1: From original code to obfuscated slices.

The protocol needs several modifications to deploy program slicing mentioned above. First two slices can be generated from the original code as shown in Figure 9.1. Program slicing is deployed before code obfuscation. The slicing criterion can be defined by MA's owner by determining a word or a set of words occurring in variables that are important (or likely to be attacked). A slicing program will generate the first slice which is related to the important variables as well as some variables randomly selected. The reason to randomly select variables is to prevent remote hosts to learn the composition of variables in a slice directly. The second slice will be related to the rest of the variables. The slices are obfuscated separately.

There will be two coordinators to handle the two slices. Each coordinator records the identity mapping of its respective obfuscated slice. Coordinator(1) with the more important obfuscated slice works similar to the coordinator in our protocol without slicing. It will generate a detector to detect tampering after the MA finishes execution on a host. Figure 9.2 shows the communications between MA and the two coordinators. Coordinator(1) needs to communicate with coordinator(2). When coordinator(1) finishes checking, if full tamper-detection is required, it orders coordinator(2)

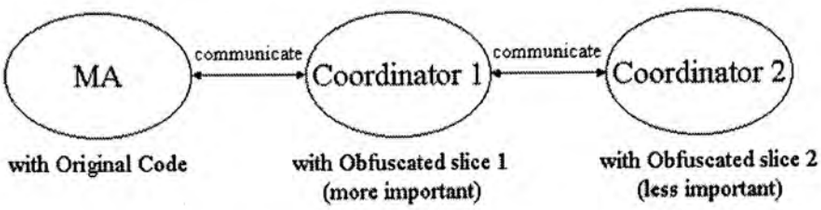

Figure 9.2: Communications between MA and coordinators with slices.

to start the detection process. And coordinator(2) will generate its detector. Coordinator(2) sends its checking result to coordinator(1). If no tampering is detected, coordinator(1) and coordinator(2) will migrate to the next host.

9.3 Conclusion

We have developed an extension to our protocol to increase detection efficiency by trading off security strength. This can be suitable in situations where MA has pre-defined trustworthy knowledge of remote hosts. The technique used is program slicing. Two or more of obfuscated slices are generated instead of a single obfuscated program. The slices are ordered by their importance (or likeliness of being attacked). The obfuscated slices perform tamper-detection one by one according to security level required. A full detection is to use all the obfuscated slices to perform detection. If a lower security level is selected for a remote host, fewer slices perform detection. Thus this can shorten detection time.

Chapter 10

Increase Scalability by Supporting Multiple Mobile Agents

10.1 Introduction

In this chapter, we present an extension to our protocol to increase scalability. There can be situations where a user sends several identical MAs to speed up performance. For example, a user may send five information seeking MAs to five separate sets of remote hosts instead of sending a single MA. In our tamper-detection protocol, a MA is monitored by a coordinator. In the example, five coordinators are needed although the code of the MAs are identical. To increase scalability of our protocol, we have developed an extension for a coordinator to support multiple MAs.

10.2 Supporting multiple mobile agents

The overhead of the protocol can be reduced by enabling a coordinator to support multiple agents. For example, a user wants to send three MAs with identical code to perform tasks on different hosts as in Figure 10.1. Instead of using three coordinators to monitor the three MAs, we can modify the protocol to enable the coordinator to support the three MAs. The obfuscated codes of the three MAs are all stored in the coordinator. Clearly the MAs should be owned by the same user or from the same home host. The coordinator can be situated on one of the honest hosts. Since the coordinator needs to use much memory resources to store MA codes and states, it should move to the honest host where memory resource is more available. The host

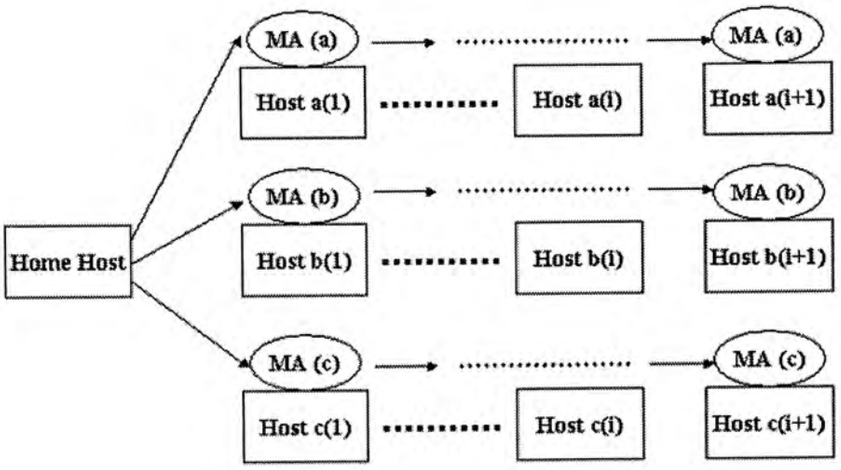

Figure 10.1: An example of itinerary of three mobile agents with identical code.

resource information can be provided by MAs (MA$\{a, b, c\}$ in Figure 10.1).

The protocol can be extended so that a coordinator can support multiple mobile agents. The prerequisite is that the codes of the multiple mobile agents must be the same. Assuming a coordinator is to support n MAs, it needs to store the states of all n MAs. It also needs to store a checking queue of hosts which has executed the agents. Compared with using n coordinators, this can save $(n-1)$ copies of the obfuscated code. However, if the idle time for the MAs is very short, it is not practical for a coordinator to support too many agents. This is because the coordinator needs time to generate detectors, communicate with detectors and check detection results. The ratio of coordinator to agent should be determined by the efficiency of tamper-detection process as well as how long a MA is idle on each host.

By comparing the method of "supporting multiple MA" to that of "supporting single MA", we can see the following advantages and disadvantages.

Advantages:

- **Only utilize one host to execute coordinator every session instead of multiple:** This is the major advantage of supporting multiple MA. Since the number of host utilized every session is reduced, host resources can be saved for other purposes. Moreover, if remote hosts have limit to number of residing agents, this method can save number of agents (coordinators) on the network. However, the host which coordinator resides needs to provide more resource since the information handled by the coordinator is increased.

- **Smaller overall migration overhead:** Since the number of coordinators is reduced to one, the number of migration process is reduced. And thus the overhead for migration is reduced compared with migration of multiple coordinators.
- **Centralized control:** The coordinator controlling all MAs can get statistics of MA status. It is possible for the coordinator to allocate tasks to the MAs if some MAs are attacked.

Disadvantages:

- **Bottleneck:** As mentioned in beginning of this section, the tamper-detection process is time-consuming. Therefore, if the MAs perform tasks very quickly, the coordinator needs to queue the host to detect into a list.
- **Single point of failure:** If the coordinator is lost or failed suddenly, all MAs will be vulnerable to tampering attack. If there are multiple coordinators, the risk can be distributed.
- **More complex control:** The control structure of this method is much more complex. This means more resource is needed for constructing the coordinator.

The major advantage of this extension is to save computation resources on remote hosts. Only one remote host is needed for handling the tamper-detection processes instead of multiple hosts. As a result, scalability of the protocol can be increased. The major disadvantage is that the remote host (supporting coordinator) becomes bottleneck and single point of failure of our tamper.

10.3 Conclusion

The extension presented in this chapter can increase scalability of the protocol. However, the extension can only work if the multiple MAs to be supported are identical in terms of code. Therefore the extension cannot be deployed in general cases.

Chapter 11

Deployment of Trust Relationship in the Protocol

11.1 Introduction

In our original protocol, we do not consider any trusted third party or trust relationship among hosts. The reason is that we want the protocol to suit open networks where trust relationships are not necessary. However, if the protocol is deployed in a network where some hosts trust others, we can speed up the protocol by deploying with the trust relationships.

11.2 Deployment of trust relationship

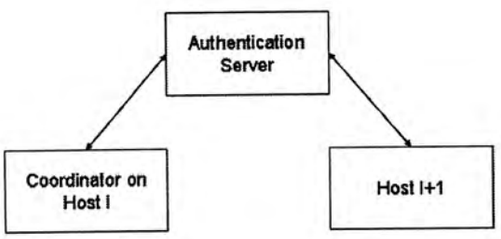

Figure 11.1: An simple authentication framework.

If a host is trusted by MA's owner (i.e. home host), we can assume that the host will not tamper with the execution of MA. Therefore, the protocol can skip the tamper-detection process on the host There must be an authentication framework in order to establish trust relationship. Figure 11.1 shows a simple authentication framework. If a host is trusted by the

MA, the coordinator needs not to check the MA's execution on the host. And the trusted host is treated as an honest host.

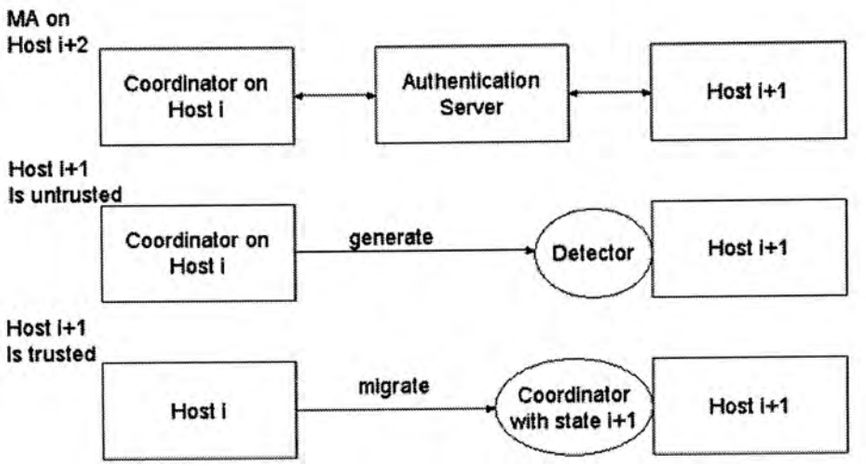

Figure 11.2: The cases for $host_{i+1}$ is untrusted and trusted.

In the protocol, the trust relationship can be deployed as in Figure 11.2. Assume MA has been executed on $host_{i+1}$ and coordinator is situated on $host_i$. After the protocol completes Step 3.1 (refer to Section 5.2), coordinator authenticates $host_{i+1}$. If $host_{i+1}$ is not trusted, the protocol works in the normal way. Otherwise, if $host_{i+1}$ is trusted, coordinator only needs to generate $state'_{i+1}$ and migrates to $host_{i+1}$. The followings show modified Step 4 to Step 6 (refer to Section 5.2) of the protocol for deploying trust relationship.

For coordinator at $host_i$:

4.1 Receive message of (arrival at $host_{i+2}$) from the mobile agent

4.2 Authenticate $host_{i+1}$

4.3 If $host_{i+1}$ is trusted by MA

- generate $state'_{i+1}$ from $state_{i+1}$ and "identity mapping table"
- send message of coordinator's (address of next host to migrate to) to the mobile agent
- migrate to $host_{i+1}$
- end of this session (goto Step 2 in next session)

4.4 If $host_{i+1}$ is not trusted by MA

- generate detector by combining detector code and obfuscated code
- load $state_i$ to obfuscated code in detector
- transfer detector to $host_{i+1}$
- goto Step 5

For detector at $host_{i+1}$:

5.1 Decrypt the message stored by the mobile agent

5.2 Execute the obfuscated code to compute $state'_{i+1}$ using $input_{i+1}$

5.3 Send $state'_{i+1}$ to coordinator

For coordinator at $host_i$:

6.1 Receive $state'_{i+1}$ from detector

6.2 Check if $state'_{i+1}$ is valid with respect to $state_{i+1}$

6.3 If result is ok (no tampering is detected),

- send message to detector to inform it to terminate
- send message of coordinator's (address of next host to migrate to) to the mobile agent
- migrate to $host_{i+1}$

6.4 If result is not ok (tampering is detected),

- send alert message to home host

To deploy trust relationship, only Step 4 of our tamper-detection protocol needs modifications. In modified Step 4, coordinator checks the trust relationship between MA and $host_{i+1}$ first. If $host_{i+1}$ is trusted, coordinator can skip tamper-detection process.

11.3 Conclusion

Our tamper-detection protocol does not require trust relationship between MA and remote hosts to protect MA. However, if MA has already established trust relationships with some remote hosts, our protocol can be extended to deploy trust relationships. Only minor modification to the protocol is needed. The modification requires coordinator to authenticate undetermined remote host before generating detector. In this way, tamper-detection to a remote host can be skipped if the host is trusted by MA. The average detection time can be speeded up.

Chapter 12

Conclusions and Future Work

Mobile agent system is an emerging paradigm in distributed systems. Mobile agent system provides three major advantages: reduced network bandwidth, customization, and asynchronous task execution. These advantages enable mobile agent systems to be particularly suitable on networks with high latency. Although the autonomy and mobility characteristics of mobile agent are attractive to applications on open networks, there are several technical problems to the paradigm. The problems include strong migration, fault tolerance, and security [18] [23]. To deploy mobile agent systems for electronic commerce applications, we have to ensure security of the systems. In particular, agent security is a new issue to traditional distributed systems. A malicious host may tamper with the mobile agent to benefit itself. To protect a mobile agent, we need to ensure its data integrity, program/code integrity, and execution/state integrity. The work described in this thesis is to detect execution tampering to mobile agents.

We have developed a tamper-detection protocol for mobile agents to ensure their execution and state integrity. Our protocol deploys three techniques: execution tracing, code obfuscation, and agent cooperation. The two kinds of cooperating agents used in the protocol are coordinator and detector. Coordinator is used to monitor the mobile agent and coordinate detection processes. Detector is used to perform re-execution process of execution tracing. The detection mechanism combines execution tracing and code obfuscation. By code obfuscation technique, the protocol can let detector to perform re-execution on undetermined hosts while coordinator always locates on honest hosts. Since honest hosts are assumed to be harmless to the mobile agent, coordinator can detect tampering in safe environment. Simultaneously, the mobile agent is involved in the detection processes, and it can perform its tasks in normal way.

Compared with existing tamper-detection protocols, our protocol has

several advantages. In Giovanni Vigna's execution tracing approach [30], tamper-detection is performed when the mobile agent goes back to home host. If the itinerary of the agent is long and the agent is tampered, the agent may have performed a number of harmful actions before going back to home host. In our protocol, detection is performed after the mobile agent finishes its execution on a remote host. This can provide faster prevention of any harmful actions of the mobile agent if it is tampered. On the other hand, Fritz Hohl's tamper-detection approach [11] is also based on execution tracing. It can prevent a tampered mobile agent to do harmful actions. However, since detection is performed before the agent's execution, the performance of the mobile agent is much affected. In our protocol, the mobile agent needs not to wait for the detection result, therefore its performance is not affected much. Moreover, the detection processes in our protocol is determined by cooperating agents. The remote hosts need not to install the detection services. Our tamper-detection protocol is more flexible to be implemented compared to Hohl's protocol, especially in old systems.

We have conducted experiments for our protocol. The experimental results show that the overheads of the protocol are acceptable. We have also verified the protocol with formal logic. And the result shows that message transmissions in the protocol are secure.

Several extensions have been developed to enhance our protocol. The first one is to solve the "fake honest host" problem of the protocol. The problem concerns about collusion of honest host and malicious host. Through the extension, the problem can be solved to a great extent with intervention of home host. The second one deploys program slicing technique to speed up the overall detection time by trading off security level. The third one enables the protocol to use a single coordinator to support several identical mobile agents. The fourth extension enables the protocol to deploy trust relationships to speed up overall detection time. Since the extensions can only be used in special cases, they are not considered as basic specification of our tamper-detection protocol.

Regarding to future work, there is still a long way to ensure security of mobile agents. First of all, work is needed to integrate the security protocols for different aspects, e.g., integrity of data, code/program, state/execution of mobile agents. An all-round security protocol is more preferable. Secondly, work is needed to integrate security protocol with fault tolerance protocol, agent locating protocol, etc. The integration of the protocols should be based on the same agent model, execution model, system model, and failure model. The Mobile Agent Facility Specification (MAFS) from Object Management Group (OMG) [22] provides a good starting point. Thirdly, since the existing

execution tracing technique can only support single-threaded program execution, further research is necessary to develop execution tracing technique which can support multi-threaded execution.

Besides, regarding to our tamper-detection protocol, it is possible to modify the protocol to suit various operating systems. In our research, we assume that a remote host can execute a mobile agent at a time. This is the safest assumption and is valid for all operating systems. However, there are operating systems which support parallel program execution (provided that the machine also support multi-processing). Professor Leung Ho-Fung has suggested that it may be possible to execute the original program and obfuscated program of the mobile agent concurrently. This is likely to reduce the delay between the end of execution and tamper-detection. To develop this extension, we firstly need to consider the execution sequences of the original program and the obfuscated program. Since the execution of the obfuscated program requires the execution trace of the original program, there are execution dependencies between program statements of the two programs. As a result, the two programs cannot be executed in parallel as separate independent programs. And the operating system may need modification to support this kind of dependent parallel execution, or the obfuscated program needs to combine with the original program to form a multi-threaded program. However, combining the obfuscated program with the original program can affect the performance of the original program. This is undesirable. Therefore, a careful study is necessary to modify our protocol to deploy parallel processing.

Appendix A

Data of Experimental Results

	Number of remote hosts				
	1	2	3	4	5
Trial 1	80	110	140	160	180
Trial 2	70	110	140	181	201
Trial 3	70	90	120	140	180
Trial 4	70	90	110	130	160
Trial 5	60	91	131	151	181
average	70	98.2	128.2	152.4	180.4

Table A.1: Single machine environment. Mesured time (in ms) for "**without** using the protocol" setting. Parameters: 1 input, 1 cycle.

	Number of remote hosts				
	1	2	3	4	5
Trial 1	100	130	160	190	220
Trial 2	91	131	161	201	221
Trial 3	100	130	160	190	210
Trial 4	100	130	170	190	230
Trial 5	100	120	151	191	211
average	98.2	128.2	160.4	192.4	218.4

Table A.2: Single machine environment. Mesured time (in ms) for "**without** using the protocol" setting. Parameters: 100 inputs, 1 cycle.

	Number of remote hosts				
	1	2	3	4	5
Trial 1	1622	2033	2464	2904	3305
Trial 2	1622	2033	2444	2864	3275
Trial 3	1632	2053	2453	2864	3275
Trial 4	1633	2053	2454	2865	3295
Trial 5	1633	2043	2464	2865	3285
average	1628.4	2043	2455.8	2872.4	3287

Table A.3: Single machine environment. Mesured time (in ms) for "**without** using the protocol" setting. Parameters: 1 input, 10000 cycles.

	Number of remote hosts				
	1	2	3	4	5
Trial 1	1653	2073	2484	2894	3325
Trial 2	1672	2083	2503	2924	3345
Trial 3	1682	2093	2504	2924	3335
Trial 4	1662	2083	2513	2934	3344
Trial 5	1652	2063	2483	2894	3344
average	1664.2	2079	2497.4	2914	3332.6

Table A.4: Single machine environment. Mesured time (in ms) for "**without** using the protocol" setting. Parameters: 100 inputs, 10000 cycles.

	Number of remote hosts				
	1	2	3	4	5
Trial 1	571	1001	1422	1442	1462
Trial 2	591	1022	1432	1452	1462
Trial 3	581	1022	1432	1442	1462
Trial 4	581	1012	1443	1453	1473
Trial 5	601	1012	1443	1453	1473
average	585	1013.8	1434.4	1448.4	1466.4

Table A.5: Single machine environment. Mesured time (in ms) for "**with** using the protocol" setting. Parameters: 1 input, 1 cycle.

	Number of remote hosts				
	1	2	3	4	5
Trial 1	1452	1492	1522	1562	1602
Trial 2	1472	1512	1552	1582	1622
Trial 3	1472	1512	1562	1602	1632
Trial 4	1482	1522	1562	1592	1632
Trial 5	1462	1502	1562	1592	1632
average	1468	1508	1552	1586	1624

Table A.6: Single machine environment. Mesured time (in ms) for "**with** using the protocol" setting. Parameters: 100 inputs, 1 cycle.

	Number of remote hosts				
	1	2	3	4	5
Trial 1	2964	3385	3785	4196	4607
Trial 2	2974	3385	3795	4206	4627
Trial 3	2964	3375	3795	4206	4617
Trial 4	3004	3395	3815	4226	4636
Trial 5	2934	3335	3755	4166	4576
average	2968	3375	3789	4200	4612.6

Table A.7: Single machine environment. Mesured time (in ms) for "**with** using the protocol" setting. Parameters: 1 input, 10000 cycles.

	Number of remote hosts				
	1	2	3	4	5
Trial 1	3025	3455	3886	4326	4747
Trial 2	3035	3485	3926	4356	4807
Trial 3	3044	3465	3895	4326	4756
Trial 4	3034	3465	3915	4356	4777
Trial 5	3064	3485	3916	4346	4777
average	3040.4	3471	3907.6	4342	4772.8

Table A.8: Single machine environment. Mesured time (in ms) for "**with** using the protocol" setting. Parameters: 100 inputs, 10000 cycles.

	Number of remote hosts		
	1	3	5
Trial 1	101	181	251
Trial 2	101	161	290
Trial 3	100	200	250
Trial 4	60	190	270
Trial 5	100	190	250
average	92.4	184.4	262.2

Table A.9: Two machines network. Mesured time (in ms) for "**without** using the protocol" setting. Parameters: 1 input, 1 cycle.

	Number of remote hosts		
	1	3	5
Trial 1	160	260	370
Trial 2	150	250	361
Trial 3	141	251	351
Trial 4	131	261	371
Trial 5	141	241	372
average	144.6	252.6	364.8

Table A.10: Two machines network. Mesured time (in ms) for "**without** using the protocol" setting. Parameters: 100 inputs, 1 cycle.

	Number of remote hosts		
	1	3	5
Trial 1	3525	5828	8152
Trial 2	3516	5839	8172
Trial 3	3465	5808	8152
Trial 4	3465	5818	8152
Trial 5	3495	5799	8162
average	3493.2	5818.4	8158

Table A.11: Two machines network. Mesured time (in ms) for "**without** using the protocol" setting. Parameters: 1 input, 10000 cycles.

	Number of remote hosts		
	1	3	5
Trial 1	3545	6119	8923
Trial 2	3755	6349	8743
Trial 3	3565	6149	8783
Trial 4	3786	6359	8933
Trial 5	3755	6359	8933
average	3681.2	6267	8863

Table A.12: Two machines network. Mesured time (in ms) for "**without** using the protocol" setting. Parameters: 100 inputs, 10000 cycles.

	Number of remote hosts		
	1	3	5
Trial 1	180	480	691
Trial 2	190	430	691
Trial 3	270	440	771
Trial 4	210	560	791
Trial 5	220	540	801
average	214	490	749

Table A.13: Two machines network. Mesured time (in ms) for "**with** using the protocol" setting. Parameters: 1 input, 1 cycle.

	Number of remote hosts		
	1	3	5
Trial 1	907	1873	2804
Trial 2	892	1843	2814
Trial 3	911	1873	2794
Trial 4	922	1933	2984
Trial 5	971	1952	2934
average	920.6	1894.8	2866

Table A.14: Two machines network. Mesured time (in ms) for "**with** using the protocol" setting. Parameters: 100 inputs, 1 cycle.

	Number of remote hosts		
	1	3	5
Trial 1	4737	8152	12277
Trial 2	4727	8192	12148
Trial 3	4657	8152	12308
Trial 4	4706	8141	12298
Trial 5	4717	8192	12197
average	4708.8	8165.8	12245.6

Table A.15: Two machines network. Mesured time (in ms) for "**with** using the protocol" setting. Parameters: 1 input, 10000 cycles.

	Number of remote hosts		
	1	3	5
Trial 1	5138	9965	13379
Trial 2	5118	9935	13319
Trial 3	5167	9834	13339
Trial 4	5148	9964	13479
Trial 5	5207	9914	13620
average	5155.6	9922.4	13427.2

Table A.16: Two machines network. Mesured time (in ms) for "**with** using the protocol" setting. Parameters: 100 inputs, 10000 cycles.

Publication

Yu Chiu-Man, Ng Kam-Wing.
A Flexible Tamper-Detection Protocol for Mobile Agents on Open Networks.
In the 2002 International Conference on Information and Knowledge Engineering (IKE'02).

Bibliography

[1] Nesria Agray, Wiebe van der Hoek, and Erik de Vink. *On BAN Logics for Industrial Security Protocols.* In B. Dunin-Keplicz and E.Nawarecki (Eds.): CEEMAS **2001**, LNAI **2296**, pp. 29-36. Springer-Verlag 2002.

[2] Shimshon Berkovits, Joshua D. Guttman and Vipin Swarup. *Authentication for Mobile Agents.* In Giovanni Vigna, editior, Mobile Agents and Security, LNCS **1419**, p. 114-136. Springer, 1998.

[3] Walter Binder and Volker Roth. *Secure Mobile Agent Systems Using Java - Where Are We Heading?* In Proc. **17**th ACM Symposium on Applied Computing, Special Track on Agents, Interactions, Mobility, and Systems (SAC/AIMS), Madrid, Spain, March 2002. ACM.

[4] M.Burrows, M.Abadi, and R. Needham. *A logic of authentication.* In ACM Operating Systems Review, **23(5)**:1-13, December 1989.

[5] David M. Chess. *Security Issues in Mobile Code Systems.* In Giovanni Vigna, editor, Mobile Agents and Security, LNCS **1419**, p.1-14. Springer, 1998.

[6] Christian Collberg, Clark Thomborson, Douglas Low. *A Taxonomy of Obfuscating Transformations.* In Technical Report **148**, Department of Computer Science, University of Auckland, July 1997.

[7] Sebastian Fischmeister, Giovanni Vigna, and Richard A. Kemmerer *Evaluating the Security of Three Java-Based Mobile Agent Systems.* In In Proceedings of Mobile Agents **2001**, volume **2240** of Lecture Notes in Computer Science, p.31-41. Springer Verlag, December 2001.

[8] Stefan Funfrocken. *Protecting Mobile Web-Commerce Agents with Smartcards.* In Proceedings of the First International Symposium on Agent Systems and Applications / Third International Symposium on Mobile Agents (ASA/MA'**99**), IEEE Computer Society, p. 90-102. 1999.

[9] Carlo Ghezzi and Giovanni Vigna. *Mobile Code Paradigm and Technologies: A Case Study.* In Kurt Rothermet, Radu Popescu-Zeletin, editors, Mobile Agents, First International Workshop, MA' **97**, Berlin, Germany, April 1997, Proceedings, LNCS **1219**, p.39-49. Springer, 1997.

[10] Fritz Hohl. *Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts.* In Giovanni Vigna, editior, Mobile Agents and Security, LNCS **1419**, p.92-113. Springer, 1998.

[11] Fritz Hohl. *A Protocol to Detect Malicious Hosts Attacks by Using Reference States.* Technical Report Nr. **09/99**, Faculty of Informatics, University of Stuttgart, Germany, 1999.

[12] Fritz Hohl and Kurt Rothermel. *A Protocol Preventing Blackbox Tests of Mobile Agents* In Fachtagung, editor, Kommunikation in Verteilten Systemen (KiVS'**99**). 1999

[13] W. Jansen, T. Karygiannis. *Mobile Agent Security.* In NIST Special Publication **800-19**. National Institute of Standards and Technology, 2000.

[14] Günter Karjoth, Danny B. Lange, and Mitsuru Oshima. *A Security Model for Aglets.* In IEEE Internet Computing, volume **1**, No. **4**, July/August 1997

[15] Danny B. Lange, Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets.* Addison Wesley, 1998. ISBN: 0-201-32582-9.

[16] Sergio Loureiro. *Mobile Code Protection.* PhD Dissertation, Instiut Eurecom, Sophia Antipolis and ENST Paris, Franc. January, 2001.

[17] Wenbo Mao and Colin Boyd. *Towards the Formal Analysis of Security Protocols.* In Proceedings of the Computer Security Foundations Workshop VI, pages 147-158. IEEE Computer Society Press, 1993.

[18] Dejan Milojicic. *Mobile Agent Applications.* IEEE Concurrency, July-September 1999 Issue, p.80-90.

[19] Dejan Milojicic, Frederick Douglis and Richard Wheeler. *Mobility on the Internet.* In Dejan S. Milojicic, Frederick Douglis, Richard Wheeler, editors, Mobility: Processes, Computers, and Agents, p. 451-456. ACM Press, February 1999.

[20] George C. Neculla and Peter Lee. *Safe, Untrusted Agents Using Proof-Carry Code.* In Giovanni Vigna, editor, Mobile Agents and Security, LNCS **1419**, p.61-91. Springer, 1998.

[21] Sau-Koon Ng and Kwok-Wai Cheung. *Protecting Mobile Agents against Malicious Hosts by Intention Spreading.* In H. Arabnia, editor, Proc.Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'**99**), Vol II, CSREA, 1999, p.725-729.

[22] Object Management Group. *Mobile Agent Facility Specification.* In "http://www.omg.org/technology/documents/formal/mobile_agent_facility.htm". 2000-01-02.

[23] A. Puliafito, O. Tomarchio, and L. Vita. *MAP: Design and Implementation of a Mobile Agent Platform.* In Journal of System Architecture. 2000.

[24] Volker Roth. *Secure Recording of Itineraries through Co-operating Agents.* In Proceedings of the ECOOP Workshop on Distributed Object Security and **4**th Workshop on Mobile Object Systems: Secure Internet Mobile Computations, p.147-154, INRIA, France, 1998.

[25] Tomas Sander and Christian F. Tschudin. *Protecting Mobile Agents Against Malicious Hosts.* In Giovanni Vigna, editor, Mobile Agents and Security, LNCS **1419**, p.44-60. Springer, 1998.

[26] Tomas Sander and Christian F. Tschudin. *Towards Mobile Cryptography.* In IEEE Symposium on Security and Privacy, May 1998.

[27] Hock Kim Tan and Luc Moreau. *Trust Relationships in a Mobile Agent System.* In Proceedings of Mobile Agents **2001**, volume **2240** of Lecture Notes in Computer Science, p.15-30. Springer Verlag, December 2001.

[28] Frank Tip. *A Survey of Program Slicing Techniques.* In Journal of Programming Languages, **3(3)**: 121-189, 1996.

[29] Giovanni Vigna. *Cryptographic Traces for Mobile Agents.* In Giovanni Vigna, editior, Mobile Agents and Security, LNCS **1419**, p.137-153. Springer, 1998.

[30] Giovanni Vigna. *Protecting Mobile Agents through Tracing.* In Proceedings of the 3rd ECOOP Workshop on Mobile Object Systems, Jyvalskyla, Finland, June 1997.

[31] Jan Vitek and Giuseppe Castagna. *Mobile Computations and Hostile Hosts.* In Journ'ees Francophones des Langages Applicatifs (JFLA**99**), p. 113-132. February 1999.

[32] White, J.E. *Telescript Technology: Mobile Agents.* General Magic White Paper, Appeared in Bradshaw, J., Software Agents, AAAI/MIT Press, 1996.

[33] Bennet S.Yee. *A Sanctuary for Mobile Agents.* In DARPA Workshop on Foundations for Secure Mobile Code, Monterey, CA, USA, March 1997. Position Paper.